MVC-malliin perustuvien Javascript-sovelluskehysten vertailua

Jouni Kähkönen

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Timo Poranen
Toukokuu 2014

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Jouni Kähkönen: MVC-malliin perustuvien Javascript-sovelluskehysten vertailua
Pro gradu -tutkielma, 76 sivua
Toukokuu 2014

Verkkosovellusten kehittämiseen on olemassa lukuisia eri Javascript-sovelluskehyksiä. Tämän tutkielman lähtökohtana on esitellä ja vertailla varteenotettavia MVC-sovelluskehyksiä keskenään. Tavoitteena on löytää kehysten laatuominaisuuksien pohjalta kuhunkin käyttötarkoitukseen parhaiten sopiva Javascript-sovelluskehys. Vertailussa on mukana kolme MVC-malliin pohjautuvaa sovelluskehystä: Maria, jQueryMX ja KnockoutJS.

Aluksi tarkastellaan verkkosovelluskehityksen taustaa ja käsitteitä. Tämän jälkeen esitellään sovelluskehysten keskeiset toiminnallisuudet. Esittelyn jälkeen kehysten ominaisuuksien pohjalta havaittavia arkkitehtuurivahvuuksia arvioidaan verkkosovellusten kehittämiseen liittyvien laatuominaisuuksien avulla. Arvioinnin jälkeen vertaillaan sovelluskehysten eroja sekä mahdollisia käyttötarkoituksia.

Yleisesti voidaan havaita, että sovelluskehystä käyttäen verkkosovelluksen ohjelmakoodin laatu paranee ja ohjelmakoodin osien välinen sidoksellisuus vähenee. Vaikka sovelluksen ulkonäkö voi olla yhtenevä kehyksestä riippumatta, vaikuttaa valinta sisäisen arkkitehtuurin myötä koodin ylläpidettävyyteen, laajennettavuuteen, tehokkuuteen ja luettavuuteen. Kehysten tarkastelun pohjalta havaittiin sopivan sovelluskehyksen valintaan vaikuttavan ainakin kolme keskeistä tekijää: Pidettäessä koodin löyhää sidoksellisuutta tärkeänä päädyttiin suosittelemaan Maria-kehyksen käyttöä. KnockoutJS-kehystä päädyttiin suosittelemaan tapauksissa, joissa sovelluksen nopeaa kehitystyötä pidetään tärkeänä. Toisaalta vaillinaisen MVC-mallin toteuttavaa jQueryMX-kehystä suositeltiin vain sovelluksiin, joissa MVC-mallin etujen tavoittelu projektin toteutuksen kannalta ei ole tärkeätä.

Kehyksen käyttö ei kuitenkaan yksinään takaa sovelluksen korkeaa laatua. Esimerkiksi suorat ristiinviittaukset sovelluksen näkymän ja mallin välillä voivat vähentää ohjelman uudelleenkäytettävyyttä toisessa projektissa. Pelkkä hyväksi havaittujen suunnittelumallien seuraaminen ei siis johda onnistuneeseen toteutukseen. Kehittäjällä on oltava harjaannusta modulaarisen ja geneerisen koodin tuottamisesta sekä selkeä näkemys sovelluksen käyttöliittymän epäkohtien havaitsemisessa.

Avainsanat ja -sanonnat: *MVC, Javascript, suunnittelumallit, sovelluskehykset.*

Sisällys

1. Johdanto

Verkkosovellusalustan luonne on muuttunut viimeisen vuosikymmenen aikana merkittävällä tavalla. Ennen sivustot olivat staattisia, ja Javascriptillä lisättiin vain hieman vuorovaikutteisuutta sivuun. Vuorovaikutteisuuden vähyyteen vaikuttivat osaltaan Javascriptin universaalin tuen puuttuminen sekä se, että Javascript oli mahdollisuuksiltaan vielä melko rajoittunut. Yleensä sivustoilla käytettiin vain lyhyitä Javascript-koodipätkiä, eikä arkkitehtuurilliseen suunnitteluun koettu olevan laajemmin tarvetta. Lisäksi sivuston suorituskykyyn pyrittiin koneiden hitaampien tehojen vuoksi kiinnittämään enemmän huomiota tekemällä Javascript-koodista toteutukseltaan mahdollisimman yksinkertaista, käyttämättä sen koommin ylimääräisiä arkkitehtuurillisia kehyksiä avuksi. Tämä heikensi auttamatta myös koodin muokattavuutta.

Tilanne palvelinpuolella on jo pitkään ollut paremmin hallussa. Arkkitehtuuriin on erilaisten sovelluskehysten avulla osattu kiinnittää huomiota aina sovelluksen kehityksen alusta asti. Asiakaspuolen koodiin eli käyttäjän selaimessa suoritettavaan ohjelmakoodiin ei välttämättä ole kuitenkaan ollut tarvetta kiinnittää huomiota, koska alkujaan koodi painottui selainpuolen koodin sijaan enemmän palvelinpuolelle, aiheuttaen sen, että selainpuolen koodista tuli usein yksinkertaista. Valitettavasti tämä käytäntö on osittain jäänyt edelleen vaikuttamaan verkkosovelluskehittäjien käytäntöihin asiakaspuolen koodin kehittämisessä.

Verkkoselain on muuttunut staattisten sivustojen esitysalustasta merkittäväksi omaksi sovellusalustakseen. Tämän myötä ohjelmakoodin määrä selainpuolen sovelluksissa on lisääntynyt, ja samalla sovelluksista on tullut sisäisesti monimutkaisempia. Näin on syntynyt tarve kiinnittää huomiota myös asiakaspuolen koodin arkkitehtuuriin.

Nyt arkkitehtuuriin panostamiseen on varaa, kun arkkitehtuurillisesta selainpohjaisesta koodista ei koidu välttämättä tuntuvaa haittaa sivuston tai verkkosovelluksen suorituskykyyn käyttäjien päätelaitteiden suorituskyvyn kasvettua. Suorituskyvyn nousun myötä on havaittavissa myös suuntausta, jossa sivuston osien asettelu keskittyy palvelinpuolen sijasta entistä enemmän selainpuolelle [Mesbah & van Deursen, 2007; Takada, 2014]. Näissä ratkaisuissa haetaan palvelimelta pelkästään raakadata, ja asiakaspuolella sijaitseva ohjelmakoodi rakentaa datan pohjalta haluamansa kaltaisen näkymän.

Verkkosovelluskehittäjät toteuttavat verkkosovelluksia usein kunakin hetkenä ratkaistavaksi haluttavan ongelman viitekehyksestä. Sovelluksen muunneltavuus ja joustavuus kärsivät, kun sovelluksen perusarkkitehtuuriin ei

ole kiinnitetty huomiota sovelluksen suunnittelu- ja kehitystyön alusta alkaen. Lisäksi ketterän kehityksen kehitysmalli saattaa johdattaa verkkosovelluskehittäjiä ajattelemaan, että itse sovelluksen sisäiseen rakenteeseen ei ole syytä kiinnittää niin paljon huomiota kuin esimerkiksi sovelluksen ensi version nopeaan valmistumiseen. Sovelluksen kehittäjä ei näin ollen ota arkkitehtuurin tai sovelluskehyksen valinnassa huomioon sitä, että lähes kaikissa sovelluksen kehittämistapauksissa asiakas asettaa myöhemmässä vaiheessa muutos- ja päivitystarpeita sovellukselleen.

Tämän tutkielman lähtökohtana on esitellä ja vertailla keskenään varteenotettavia MVC-sovelluskehyksiä ja pyrkiä löytämään sovelluskehysten laatuominaisuuksien pohjalta sopivin sovelluskehys Javascript-sovelluskehittäjien käyttöön. Erityinen painopiste vertailussa on kehyksen kyvyssä eriyttää datan käsittely, näkymä ja ohjelmalogiikka erillisiksi, mahdollisimman itsenäisesti toimiviksi kokonaisuuksiksi.

Kehyksetön Javascript-koodaus, tarkoitukseen sopimattoman kehyksen käyttö tai kehyksen käyttö väärään tarkoitukseen saattaa johtaa koodiin, joka on siirrettävyydeltään, muokattavuudeltaan ja ymmärrettävyydeltään heikkoa. Toisaalta myös oikeantyyppisen Javascript-sovelluskehyksen löytäminen voi olla vaikeaa, joten on otettava huomioon kehysten eri painopisteet ja laatuominaisuudet, jotta kuhunkin tilanteeseen parhaiten sopiva sovelluskehys voidaan löytää.

Luvussa 2 esittelen verkkosovellusten ohjelmistokehityksen taustaa sekä keskeisimmät selainpohjaisen ohjelmistokehityksen käsitteet. Selvennän myös, miten olio-ohjelmoinnista tutut käsitteet vastaavat selainsovelluksissa käytetyn Javascript-kielen prototyyppipohjaisia käsitteitä. Luvussa 3 kerron yleisesti sovelluskehyksistä, niiden eduista ja haitoista. Jaottelen myös selainpohjaiset sovelluskehykset eri sovelluskehystyyppeihin sekä esittelen sovelluskehysten vertailussa huomioitavat laatuominaisuudet. Luvussa 4 esittelen kolme verkkosovelluksien kehittämiseen käytettyä sovelluskehystä. Tuon esiin sovelluskehyksen MVC-mallin sovellustavan, kehyksen tyypillisimmät käyttötavat sekä omintakeisimmat sovelluskehyksessä esiintyvät piirteet. MVC-mallin käyttöä havainnollistan lyhyin koodiesimerkein.

Luvussa 5 arvioin ja vertailen esitettyjä sovelluskehyksiä painottaen enemmän kehyksellä tuotettavien sovellusten koodirakenteellisia piirteitä kuin kehyksen omaa, sisäistä rakennetta. Lopuksi esitän yhteenvedon siitä, mitkä ovat keskeisimmät erot sovelluskehysten välillä ja mikä sovelluskehys sopii parhaiten kuhunkin käyttötarkoitukseen.

2. Verkkosovellusten ohjelmistokehitys

Ohjelmistokehityksessä on usein kyse havaitun ongelman ratkaisemiseen tarkoitetun tietokoneohjelman suunnittelusta ja toteutuksesta. Verkkosovellusten ohjelmistokehitys rajautuu käsitteenä verkkoyhteyttä hyödyntävien sovellusten kehittämiseen. Tässä tutkielmassa ohjelmistokehityksen viitekehys rajautuu vielä verkkoselaimessa ajettavien sovellusten ohjelmistokehityksen tarkasteluun ja erityisesti verkkoselainpohjaisten sovellusten kehittämiseen käytettävien sovelluskehysten arkkitehtuurin arviointiin.

Tässä luvussa kuvaan verkkosovellusten ohjelmistokehityksen ominaispiirteitä sekä keskeisiä verkkosovellusten kehittämisessä ilmeneviä käsitteitä. Kohdassa 2.1 käsittelen verkkosovellusten luonnetta, verkkoalustan yleistä arkkitehtuuria sekä verkko- ja työpöytäsovellusten eroja. Lisäksi pyrin luomaan kuvaa sovellusalustassa tapahtuneesta kehityksestä viimeisen vuosikymmenen aikana. Tämän jälkeen esittelen keskeisimmät selainpohjaisessa ohjelmistokehityksessä esiintyvät käsitteet (kohta 2.2), verkkosovelluksissa käytettävät MVC-mallit (kohta 2.3) ja suunnittelumallit (kohta 2.4) sekä lopuksi oliomaailman käsitteitä vastaavat Javascript-kielen tekniikat (kohta 2.5).

2.1. Yleistä verkkosovellusten kehityksestä

Verkkosovellusten ohjelmistokehitys pohjautuu perimmiltään palvelimen ja asiakkaan väliseen kommunikointiin. Lisäksi verkkosovellusten ohjelmistokehityksessä on kiinnitettävä erityistä huomiota sovelluksen tietoturvaan, sillä palvelin joutuu vastaanottamaan asiakkaalta dataa ja soveltamaan sitä käyttötarkoituksen edellyttämällä tavalla.

Palvelimen ja asiakkaan välisessä kommunikoinnissa, niin kutsutussa *palvelin–asiakas*-suhteessa, osa sovelluskoodista jakautuu palvelimelle ja osa käyttäjän selaimella suoritettavaksi. Lisäksi sivuston tai sovelluksen sisältö rakennetaan osin tai kokonaan palvelinpuolella, minkä jälkeen sisältö lähetetään käyttäjän selaimelle. Palvelin–asiakas-malli mahdollistaa myös tarpeen mukaisen tiedon kätkennän: verkkosovellusten ohjelmistokehityksessä saatetaan jättää tärkeää sovelluskoodia, esimerkiksi yritykselle arvokas hintalaskufunktio, tarkoituksella vain palvelimella suoritettavaksi, sillä palvelimella sijaitseva koodi ei ole selaimesta käsin käytettävissä.

Keskeiseen asemaan verkkosovellusten ohjelmistokehityksessä nousee palvelimella sijaitsevan koodin tietoturva. Mikään ei estä sovelluksen käyttäjää muokkaamasta selaimeen tuodun lähdekoodin sisältöä itselleen edullisella tavalla. Tämän takia palvelimella sijaitsevassa koodissa ei tulisi sokeasti luottaa

selaimelta tulevaan dataan, vaan kaikki tieto tulisi pääsääntöisesti *sanitoida* tilanteen edellyttämällä tavalla. Vaikka verkkosovellusten ohjelmistokehityksessä tiedon turvallisuuden tarkistuksen tarpeellisuus tiedetään jo laajasti, kuulee jatkuvasti uutisia verkkosovelluksiin kohdistuneista hyökkäyksistä, jotka johtuvat nimenomaan palvelimen vastaanottaman tiedon turvallisuuden riittämättömästä tarkistuksesta.

2.1.1. Verkko- ja työpöytäsovellusten eroja

Verkkosovellusten ohjelmistokehitys eroaa jossain määrin työpöytäsovelluksiin liittyvästä ohjelmistokehityksestä. Verkkosovelluksissa alustana on käyttäjän selain, työpöytäsovelluksissa käyttöjärjestelmä. Useimmissa tapauksissa työpöytäsovellukset toimivat vain sillä alustalla, jolle ne on suunniteltu. Verkkosovellusten etuna on se, että ne ovat käytettävissä käyttäjän käyttöjärjestelmästä riippumatta, millä tahansa tuetulla selainalustalla. Toisaalta työpöytäsovellukset ovat usein suorituskyvyltään tehokkaampia, sillä sovelluksia voidaan lähes aina ajaa natiivisti suoraan tietokoneen suorittimella. [Bychkov, 2013]

Työpöytäsovellusten ohjelmistokehitys eroaa myös asennustavaltaan ja päivitettävyydeltään verkkosovelluksista. Työpöytäohjelma on yleensä kehitettävä tiettyä käyttöjärjestelmää ja laitteistoa varten, ja se on niin ikään asennettava käyttäjän käyttöjärjestelmään. Ohjelmistoon tulevat päivitykset on myös asennettava jokaiseen asennukseen erikseen, kun taas verkkosovelluksissa päivitys joudutaan usein tekemään vain yhteen, palvelimella sijaitsevaan sovelluksen asennukseen. [Bychkov, 2013]

Toisaalta verkkosovellukset näyttäisivät olevan ylläpidettävyydeltään häiriöalttiimpia kuin työpöytäsovellukset. Verkkosovelluksen käyttäjä on riippuvainen verkkosovelluksen palvelimen toimintavarmuudesta. Jos verkkopalvelin, jolle sovellus on asennettu, murretaan tai verkkopalvelin jumiutuu, mahdollinen käyttökatkos voi vaikuttaa kaikkiin kyseisen sovelluksen käyttäjiin. Tätä vaaraa ei ole verkkoyhteydestä riippumattomissa työpöytäsovelluksissa.

Alkujaan tietokoneohjelmat toimivat suurten keskustietokoneiden alaisuudessa. Asiakaskoneet olivat vähätehoisia päätteitä, jotka käyttivät keskustietokoneen tehoja tiedonkäsittelyyn. Tietokoneiden suoritintehojen ja muistimäärän kasvettua alettiin tehdä yhä enemmän käyttäjän tietokoneen suoritintehoja ja muistikapasiteettia hyödyntäviä sovellusohjelmia. [Bychkov, 2013] Vastaavanlaista kehityskulkua näyttää esiintyvän myös verkkosovellusten ohjelmistokehityksessä; verkkosovellusten toimintalogiikka ei painotu nykyään enää niin paljon palvelinpuolelle, vaan sovelluksen käyttöliittymään ja tiedon

käsittelyyn liittyviä toimintoja suoritetaan yhä enemmän käyttäjän selaimessa, käyttäjän koneen resursseja hyödyntäen.

Nykyään työpöytä- ja verkkosovellusten ero näyttää hieman kaventuneen. Verkkosovelluksia on mahdollista integroida yhä enemmän käyttäjän työpöytään: HTML5-standardin ja Javascript-kielen kehittyneiden tekniikoiden ansiosta verkkosovelluksiin pystyy esimerkiksi raahaamaan tietoa suoraan käyttäjän työpöydältä [Mozilla, 2014a]. Lisäksi on jo olemassa sovellusrajapintoja, joiden kautta verkkosovellus voi luvan saatuaan päästä käsiksi käyttäjän tiedostojärjestelmään [Html5rocks.com, 2014]. Tietyin tekniikoin on myös mahdollista ajaa verkkosovelluksia ilman selainta, omassa ikkunassaan [Sopyło, 2013; Adobe, 2014].

Lisäksi suorituskykyerot työpöytä- ja verkkosovellusten välillä ovat kavenneet. Alkujaan vain työpöytäsovelluksia ja muita paikallisia sovellusohjelmia ajettiin tietokoneen suorittimella natiivisti, mutta nykyisten kehittyneiden selainten komentosarjaesikäännöstekniikan ansiosta on niin ikään mahdollista päästä selainpohjaisilla sovelluksilla lähes natiivien sovellusten suoritustehotasolle.

2.1.2. Palvelinkeskeisyydestä asiakaskeskeisyyteen

Ensimmäiset MVC-pohjaiset verkkosovelluskehykset keskittyivät usein niin sanottuun *kevyeen asiakaspäätteeseen* (engl. *thin client*), jossa lähes kaikki mallin, näkymän ja ohjaimen logiikka sijoitettiin palvelinpuolelle. Tällaista tekniikkaa käytettäessä käyttäjä lähettää joko hyperlinkkipyynnön tai lomakkeen syötteen selainpuolelta palvelinpuolen ohjaimelle, minkä jälkeen palvelin lähettää näkymänsä avulla selaimelle kokonaan päivitetyn verkkosivun. Selainpuolen ohjelmistotekniikoiden kehityttyä on tehty erilaisia selainpuolen sovelluskehyksiä, joiden avulla MVC-sovelluksen vastuualueet voidaan osittain tai kokonaan suorittaa asiakaspuolella. [Leff & Rayfield, 2001]

Ohjelman toimintalogiikkaan liittyvän koodin painottuminen asiakaspuolelle on johtanut toimintojen vasteajan nopeutumiseen, kun sivuston muutosten näyttämiseksi käyttäjälle ei vaadita sivuston lataamista palvelimelta. Lisäksi vasteaikaa saadaan nopeutettua, vaikka tietoja jouduttaisiinkin lataamaan palvelimelta, sillä nykyaikaisten verkkosovellusten tekniikoilla palvelimelta voidaan ladata vain muuttunut osa sivustosta, tarvitsematta päivittää koko sivustoa uudelleen.

2.1.3. Sovellusalustan kehittyminen

Vuosituhannen vaihteen alussa verkkosovellusten toteuttamisessa nojauduttiin vielä pitkälti selainlaajennusten tarjoamaan ohjelmistorajapintaan. Sovelluksia tehtiin paljon muun muassa Adobe Flash Player -selainlaajennuksen pohjalle. Lisäksi useat yritykset käyttivät selaimen laajennuksena toimivaa Java Applet -tekniikkaa sovellusratkaisuissaan. Tästä suuntauksesta aiheutui lukuisia sovelluksen käytettävyyteen ja siirrettävyyteen liittyviä ongelmia: verkkosovelluksen toimivuus oli kiinni selaimeen asennettavasta lisäosasta, jolloin sovelluksen käyttö esimerkiksi mobiililaitteilla tai vapaan lähdekoodin käyttöjärjestelmissä ei aina ollut mahdollista. Etenkin kannettavien laitteiden tehot eivät taanneet riittävän tehokasta Flash-suorituskykyä, tai laitteisiin ei sisällytetty ollenkaan tukea Flash-teknologialle sen hitaamman suorituskyvyn takia.

Viimeisten vuosien aikana tällä saralla on kuitenkin havaittu positiivista kehitystä. Verkkosovellusten toteuttaminen selainlaajennusten varaan on huomattavasti vähentynyt. Tähän on merkittävästi vaikuttanut se, että suuret tietotekniikkayhtiöt ovat pyrkineet vähentämään ulkoisiin sovelluslaajennuksiin nojautuvaa sovellusten toteuttamiskehityssuuntaa ja sen sijaan olleet mukana HTML5-tekniikan eteenpäin viemisessä. Esimerkiksi Apple jätti kokonaan sisällyttämättä Flash-tuen matkapuhelimiinsa ja muihin mobiililaitteisiinsa muun muassa suorituskykyyn ja tietoturvaan liittyvin perustein [Jobs, 2010]. Näin verkkosivustojen toteuttajien on ollut pakko toteuttaa selainlaajennuksiin nojautuvat sivustotoiminnallisuudet uudelleen siten, että sivustot käyttävät natiivia HTML5-tekniikkaa selainlaajennusten asemesta.

Tämä kehityssuunta vahvistaa HTML5-pohjaisten verkkosovellusten merkitystä. Tämän myötä myös sopivan Javascript-sovelluskehyksen valitsemisen tärkeys korostuu, kun tarjolla on lukuisia erilaisia sovelluskehyksiä erilaisine arkkitehtuuriratkaisuineen. Ongelmana on silti valinnan vaikeus: mikä kehys soveltuisi arkkitehtuuriltaan ja muilta ominaisuuksiltaan parhaiten kulloinkin suunnitteilla olevan sovelluksen toteutukseen. Lisäksi esiin nousee kysymys, kannattaako sovellus tehdä jopa ilman sovelluskehystä vai voiko kyseeseen tulla peräti oman yksinkertaisen sovelluskehyksen toteuttaminen. Ongelma on yhä ilmeisempi, kun yritykset vaativat tuoteratkaisuissaan työpöytäsovellusten sijaan enemmän selaimeen perustuvia käyttöliittymiä.

2.2. Käsitteet

Tässä kohdassa esittelen keskeisimmät selainpohjaisessa ohjelmistokehityksessä esiintyvät käsitteet sekä merkittävät asiaa sivuavat tekniikat. Aluksi puhutaan yleisesti verkkosovelluskehitykseen liittyvistä kielistä ja tiedonsiirto-

muodoista. Sen jälkeen käsitellään sovellusten kehityksessä käytettäviä yhteyskäytäntöjä. Lopuksi tuodaan myös käyttöliittymää koskevat keskeiset käsitteet esiin, koska verkkosovelluksissa on loppujen lopuksi kyse käyttöliittymien rakentamisesta.

2.2.1. Kielet ja tiedonsiirtomuodot

HTML-merkkauskielellä (engl. *Hypertext Markup Language*) esitetään verkkosivujen ja -sovellusten rakenne. Tässä hypertekstin luomiseen käytettävässä merkkauskielessä dokumentin rakenne kuvataan *elementteinä*, joiden sisällä voi olla toisia elementtejä. Elementtien ominaisuuksia voi tarkentaa elementtien *tunnisteiden* (engl. *tag*) sisällä määritetyin *määrein* (engl. *attribute*). Dokumentin rakenteesta voidaan viitata muihin dokumentteihin tai resursseihin käyttäen *URI*-viitteitä (engl. *Uniform Resource Identifier*) eli verkko-osoitteita. [W3C, 1999; Berners-Lee, 1989]

Tiedon rakenteen ja sisällön esittämiseen ja siirtämiseen käytetään usein *XML*-kuvauskieltä (engl. *Extensible Markup Language).* Se on läheistä sukua HTML-merkkauskielelle, mutta sillä on eri käyttötarkoituksensa: uusia elementtejä voi määrittää ilman rajoituksia, ja elementeille voi antaa datan rakenteen asettamien vaatimusten mukaisia merkityksiä. XML:ssä tiedon rakenteen oikeellisuus on tarkempaa; esimerkiksi tunniste on aina päätettävä lopputunnistetta käyttäen, kun taas HTML-kielessä tietyt tunnisteet eivät vaadi lopputunnistetta. [W3C, 2008]

XML on ominaisuuksiltaan monipuolinen merkkauskieli, mutta jossain mielessä myös hieman *verboosi* eli monisanainen: merkkaus voi viedä jopa enemmän tilaa kuin dokumentissa esitettävä data. Tämän ja muiden seikkojen takia on kehitetty muita, yksinkertaisia verkkosovellustekniikkoihin liittyviä tiedonkuvauskieliä.

Eräs tällainen kieli on *JSON* (engl. *Javascript object notation* eli *Javascript-oliomerkintätapa*). Se kuuluu nykyään keskeisimpiin selainsovelluskehityksessä käytettäviin tiedonkuvaustapoihin. Tiedostomuodon kehittäjien mukaan kieltä on ihmisen helppo lukea ja tuottaa, mutta samalla koneiden helppo jäsennellä ja generoida. Sen avulla tietoa voidaan siirtää XML-kieltä yksinkertaisemmassa muodossa. Tallennusmuoto muistuttaa paljolti assosiatiivista tietotaulukkoa, sillä se koostuu kahdesta tietorakenteesta: a) luettelosta nimi-arvo-pareja, jotka ovat rinnastettavissa useissa kielissä olion käsitteeseen sekä b) järjestetystä listasta, joka muistuttaa perinteistä taulukkoa (engl. *array*). [JSON, 2014]

Vaikka merkintätavan nimessä puhutaan oliomerkinnästä, JSON-muotoiseen dataan ei voi sellaisenaan tallentaa olioita. Oliot, kuten jopa yksinkertaiset

päivämäärätietueet, joudutaan sarjallistamaan merkkijonomuotoon, jotta tiedon tallennus JSON-muodossa onnistuu. Toisaalta tätä voinee pitää JSON-merkintätavan vahvuutena, sillä näin kyseisessä muodossa tallennettu tieto on kieliriippumatonta.

HTML-dokumenttiin voi sisältyä myös selaimessa ajettavaa ohjelmakoodia. Tällöin kyse ei välttämättä enää ole dokumentista vaan selainpohjaisesta verkkosovelluksesta. Ohjelmakoodi voidaan liittää suoraan dokumentin rakenteeseen *script*-tunnisteella, tai koodi voidaan sisällyttää dokumenttiin viittaamalla erilliseen resurssiin. Yleisin verkkosovellusten selainpuolen ohjelmointikieli on *Javascript*, josta kerrotaan lisää kohdassa 2.5.

Minimoinnilla (engl. *minification*) sovelluskehyksen koodi saadaan toimitettua usein yhtenä lyhennettynä komentosarjatiedostona, minkä ansiosta se saadaan ladattua nopeammin käyttäjän selaimeen verkkoyhteyden välityksellä. Minimoidusta lähdekoodista on poistettu tarpeettomat välilyönnit, rivinvaihdot ja sarkainmerkit sekä koodikommentit. Usein myös useampi komentosarjatiedosto on liitetty yhteen minimoituun komentosarjatiedostoon, sillä yhden tiedoston lataus vaatii useamman sijasta vain yhden http-pyynnön, mikä nopeuttaa sivuston latautumista entisestään. [Sahgal, 2014a]

Minimointia ei tule sekoittaa *obfuskointiin* [JavaEncrypt, 2011] eli koodin monimutkaistamistekniikkaan, jossa varjeltavan lähdekoodin sisältö tarkoituksella sekoitetaan lukukelvottomaan muotoon [Sahgal, 2014b]. Javascript tulkattavana kielenä on otollinen minimoinnin käytölle. Koodia ei muunneta tavukoodi- tai binäärimuotoon, joten lähdekoodin koolla on merkitystä tiedonsiirron tehokkuudessa. Minimointi ei rajoitu vain ohjelmakoodiin; minimoida voi myös HTML- ja CSS-muotoista dataa.

Keskeisenä käsitteenä ohjelmakoodin kannalta on *DOM*-ohjelmistorajapinta (engl. *Document Object Model*). Dokumenttioliomallin avulla selainpohjainen ohjelma pääsee käyttämään ja muokkaamaan HTML- ja XML-dokumenttien rakennetta. Rajapinta määrittelee tavan dokumentin tietojen hakemiseen ja muokkaamiseen. Dokumentti, niin sanottu *DOM-puu*, esitetään *solmuina*, joiden kautta komentosarjasta pääsee käsiksi dokumentin rakenteeseen. [Peltomäki & Nykänen, 2006]

Ilman DOM-rajapintaa sovelluksen käyttöliittymäelementtien käyttö olisi hyvin vaikeaa, koska sovelluskehittäjän pitäisi itse jäsentää HTML-dokumentin sisältöä merkkijonodatana. On tärkeää huomata, että selainsovelluksissa DOM-rajapinnan kautta käytettävä dokumentti vastaa sovelluksen näkymän rakennetta. Nykyään DOM-rakenteen muokkaamista voi helpottaa Javascript-kirjastoilla, kuten *jQuery* [jQuery, 2014].

2.2.2. Verkkosovelluskehityksen yhteyskäytännöt

Yhteyskäytäntö on sovittu standardi laitteiden tai sovellusten välisessä yhteydessä käytettävistä keskustelusäännöistä. Yhteyskäytännön avulla mahdollistuu viestien lähettäminen osapuolten välillä. Verkkosovellusten tapauksessa käytetään niin kutsuttua *http*-yhteyskäytäntöä, johon kuuluu keskeisenä *asiakkaan* ja *palvelimen* eriytetty rooli. Yhteyskäytännön avulla sivusto voidaan ladata kokonaisuudessaan palvelimelta asiakkaalle, tai asiakas voi ladata dataa palvelimelta osittain, päivittämättä koko sivustoa.

Asiakas on palvelimelta tulevan tiedon pyytäjä tai vastaanottaja, tai palvelimelle tiedon lähettävä osapuoli. Asiakkaana on usein selain, mutta se voi yhtä lailla olla jokin muu sovellus, joka käyttää palvelimen rajapintaa. Tässä asiakkaaseen viitataan selaimen lisäksi myös termillä *selainpuoli*.

Pyynnöllä selain käskee palvelinta lähettämään resurssin sisällön tai muuttamaan resurssin sisältöä. Pyynnön mukana annetaan haettavan tiedon osoite eli *URI*-osoite, käytettävä metodi eli niin kutsuttu *verbi*, joita voi olla muun muassa *GET*, *POST* ja *DELETE* sekä mahdollisesti palvelimelle lähetettävä datan sisältö ja tyyppi. Pyyntö koostuu otsakkeesta (engl. *header*) ja runko-osasta (engl. *body*). Otsakkeessa on pyyntöön liittyvää oheistietoa, kuten käyttöjärjestelmäversio, selainversio ja viittaavan sivun osoite. [RFC2616, 1999]

Palvelin vastaa pyyntöön vastaavanlaisella yhteyskäytännöllä. Vastauksen otsakkeessa on palvelimeen liittyviä tietoja, kuten palvelimen käyttöjärjestelmä ja versio sekä tiedon sisältöön liittyviä tietoja, kuten tyyppi, merkistökoodaus ja tilakoodi. Vastauksen runko sisältää pyydetyn resurssin tietosisällön, joka koostuu esimerkiksi verkkosivujen tapauksessa dokumentin HTML-rakenteesta. Runko voi puuttua vastauksesta, esimerkiksi palvelimen antaessa uudelleenohjauspyynnön. [RFC2616, 1999]

Ajax-tekniikan (engl. *Asynchronous Javascript and XML*) avulla selainpohjaisen sovelluksen koodi voi hakea ja lähettää tietoja palvelimen ja selaimen välillä ilman koko sivuston uudelleenlataamista. Ajax-tekniikan avulla verkkosivujen vuorovaikutteisuutta voi lisätä. XML-muotoisen datan sijasta selain voi lähettää minkä tahansa muun tyyppistä dataa, kuten JSON-dataa. [Peltomäki & Nykänen, 2006]

Tekniikkana Ajax on hyvin yksinkertainen; se ei sisäisesti paljon eroa normaalista http-pyynnöstä. Se kuitenkin avaa selainpohjaisille verkkosovelluksille lukemattomia mahdollisuuksia verkkosovelluksen vuorovaikutteisuuden lisäämiseen. Yleisellä tasolla kuvattuna Ajax-tekniikka mahdollistaa siirtymisen sivustomallista varsinaiseen sovellusohjelmamalliin, jolloin saavutetaan työpöytäsovellusta muistuttava käyttökokemus [Agile, 2013].

2.2.3. Käyttöliittymän käsitteet

*CSS (*engl. *Cascading Style Sheets)* on merkkauskieli, jolla HTML-dokumentin elementteihin voidaan liittää tyylimäärittelyjä. CSS-kielen avulla sivuston tyylimäärittelyt saadaan erotettua HTML-dokumentin rakenteesta. Lisäksi samaa tyylimäärittelyä voidaan soveltaa useassa eri elementissä, mikä niin ikään lisää sivuston modulaarisuutta. [W3C, 2011]

CSS on melko joustava ja monipuolinen merkkauskieli tyylien määrittelyyn, mutta sillä on heikkoutensa: Jos saman elementin tyylimäärittelyn alle määritetään muita tyylimäärittelyjä, joudutaan ulompi elementti määrittämään jokaisen alempana määritellyn tyylimäärittelyn yhteydessä uudestaan.

Tämän ja lukuisten muiden CSS-kielen ongelmien korjaamiseksi on kehitetty useita kolmannen osapuolen tyylimäärittelykieliä, jotka yksinkertaistavat muun muassa sisäkkäisten tyylimäärittelyjen käyttöä. Nämä kielet kääntyvät joko ajon aikana tai ennen ajoa selaimen ymmärtämään CSS-muotoon. Esimerkkejä tällaisista käännettävistä tyylimäärittelykielistä ovat *LESS* [lesscss.org, 2014] ja *Sass* [Sass-lang.com, 2014].

Käytettävästä kehyksestä riippumatta verkkosovelluksen käyttöliittymät liitetään sovelluksen ohjelmakoodiin *käyttöliittymätapahtumien* avulla. Tapahtuma on välikappale käyttöliittymän ja sovelluksen toimintalogiikan välillä: järjestelmä kutsuu tiettyä sovelluksen funktiota tai metodia käyttäjän syötteen tai muun järjestelmän tapahtuman myötävaikutuksesta.

Verkkosovelluksissa tapahtumafunktiolle annetaan parametrina usein *event*-olio, josta pääsee käsiksi tapahtumaan liittyvään käyttöliittymäelementtiin. Olion kautta saa myös tietoja tapahtumasta, kuten painetusta näppäimestä tai hiiren osoittimen sijainnista napsautuksen aikana. [Peltomäki & Nykänen, 2006]

2.3. MVC-malli ja sen johdannaiset

Olennainen osa sovelluskehyspohjaista ohjelmistokehitystä on MVC-mallin käyttö. MVC-suunnittelumallin mukaan toteutettu sovellus eriyttää tiedonkäsittelyn, graafisen näkymän sekä ohjelmalogiikan toiminnallisuudet toisistaan. Tavoitteena on erottaa eri vastuualueet toisistaan, usein omissa luokissaan käsiteltäviksi, jotta ohjelman osa-alueiden riippuvuus toisistaan saadaan minimoitua.

MVC-mallia ja sen johdannaisia kutsutaan *esitysmalleiksi* (engl. *presentation pattern*) [Manoj, 2012a]. Niiden avulla ohjelman tilan esitys saadaan eriytettyä ohjelman muusta toiminnasta. Useat esitysmallit ovat muunnoksia alkuperäisestä MVC-mallista.

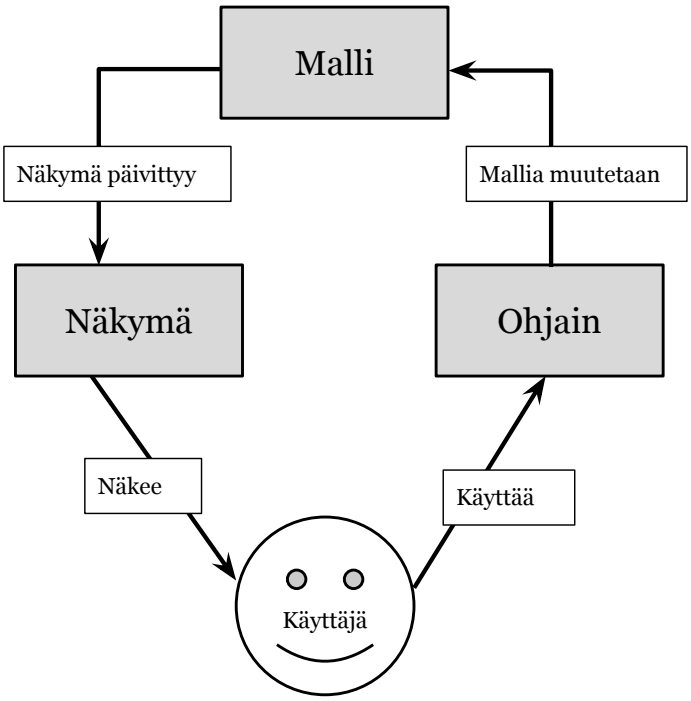
Suunnittelumalleista ja MVC-johdannaisten malleista puhuttaessa on sekaannuksen välttämiseksi syytä mainita aiheeseen liittyvä suomenkielisen termistön monitulkintaisuus. Englannin kielessä suunnittelumallista (engl. *design pattern*) ja MVC:n mallista (engl. *model*) käytetään eri nimitystä. Tässä tutkielmassa *MVC-mallilla* tarkoitetaan nimenomaan MVC-suunnittelumallia, ei MVC-suunnittelumallin tiedonkäsittelystä vastaavaa *mallia*.

Nykyisten verkkosovelluskehysten voidaan katsoa päällisin puolin toteuttavan MVC-mallin periaatteet. Kehysten MVC-toteutuksissa on kuitenkin paljon eroja ja jopa selvästi havaittavia puutteita. Jotkin kehykset käyttävät MVC-mallin asemesta jotain muuta esitysmallia, kun taas joidenkin kehysten kehittäjät väittävät kehyksensä toteuttavan MVC-mallin periaatteen, vaikka kehystä ei voitaisi selvästi kategorisoida mihinkään olemassa olevista MVC-johdannaisista.

Esimerkiksi Backbone.js-kehys [Backbone.js, 2014] käyttää lähinnä MVP-suunnittelumallia tai jonkin asteen muunnosta tästä. Kehyksessä ohjaimen tilalla on niin sanottu esittäjä (engl. *presenter*), jolloin ohjaimen käytön sijasta ohjelmalogiikkaan liittyvä ohjelmakoodi siirtyy näkymän harteille. Esittäjänä kehyksen esitysmallitoteutuksessa toimii näkymään viittaava View-olio [Backbone.js, 2014], jolloin ohjaimen ja näkymän rooleja ei ole selkeästi erotettu. Osmani [2012b] toteaa, ettei kyseisen kehyksen arkkitehtuuri vastaa niin MVC-, MVP- kuin MVVM-mallia, vaan kehyksessä käytetään omaa, MV*-perhettä muistuttavaa arkkitehtuurilähestymistapaa.

2.3.1. MVC

MVC-mallissa sovellus tai sen yksittäiset osat jaetaan malliin, näkymään ja ohjaimeen. MVC-mallin idean kehitti Trygve Reenskaug vuonna 1979 Smalltalk-ohjelmointikielelle. Alkuperäisen kuvauksen [MVC-Process, 2010] pohjalta uudelleenpiirretty kuva esittää tyypillisen mallin, ohjaimen, näkymän ja käyttäjän välisen vuorovaikutuksen (kuva 1). Käyttäjä vastaanottaa visuaalisia kuvia näkymältä, käyttää käyttöliittymässä esitettyjä ohjaimen toimintoja, jolloin ohjain muuttaa toiminnon edellyttämällä tavalla mallin tilaa. Malli vuorostaan tiedottaa tilansa muuttumisesta näkymälle (ja mahdollisesti muille mallin tilaa tarkkaileville olioille), jolloin näkymä päivittää käyttöliittymänsä vastaamaan mallin uutta tilaa. [Manoj, 2012a]



Kuva 1. Tyypillinen MVC-kiertokulku mallin, ohjaimen, näkymän ja käyttäjän vuorovaikutuksen välillä. [MVC-Process, 2010]

Ohjain (engl. *controller*) vastaanottaa käyttäjän syötteen ja pyytää mallia hakemaan tai tallentamaan syötettä vastaavan datan. Syötteenä voi olla esimerkiksi painikkeen napsautus, tekstikentän muuttaminen tai mikä tahansa muu käyttäjän toiminto. Jos MVC-toteutus ei käytä tarkkailijamallia, ohjain itse pyytää näkymää esittämään datan. Tarkkailijamallia käyttävässä toteutuksessa mallin muutos aiheuttaa näkymän päivityksen. Ohjainta kutsutaan suomen kielessä myös käsittelijäksi tai kontrolleriksi. [Reenskaug, 1979; Manoj, 2012a]

Malli (engl. *model*) sisältää sovelluksen datan tai on välillisesti yhteydessä sovelluksen dataan. Malli on usein passiivinen toimija; se hakee tai tallentaa dataa vain, kun ohjain (tai joissain toteutuksissa suoraan näkymä) käskee niin. Malli voi sisältää datan itsessään tai hakea sen jostain muusta lähteestä, kuten erillisestä tietokannasta. Kun mallin tila muuttuu, se käskee mallia tarkkailevia olioita, kuten näkymää, päivittämään tilansa muutoksen mukaisesti. [Reenskaug, 1979; Manoj, 2012a]

Näkymä (engl. *view*) rakentaa käyttöliittymän esittäen suoraan mallilta tai ohjaimen välityksellä saadun datan käyttäjän ymmärtämässä, graafisessa muodossa [Reenskaug, 1979; Manoj, 2012a]. Lisäksi näkymä päivittää käyttöliittymää mallin muutosten mukaisesti sekä joissain tapauksissa erikseen ohjaimen käskystä. Joissakin verkkopohjaisissa MVC-ratkaisuissa näkymäksi kutsutaan HTML-dokumentin DOM-rakennetta, kun taas joissain ratkaisuissa DOM-rakennetta muokataan erillisen näkymäluokan (tai erillisten näkymäluokkien) kautta.

Reenskaug itse kuvaa mallin luontia seuraavasti: "MVC-mallista suunniteltiin yleinen ratkaisu ongelmaan, jossa käyttäjät joutuvat ohjaamaan suurta ja monimutkaista tietomäärää. Vaikeinta oli keksiä sopiva nimi arkkitehtuurin eri komponenteille. Model-View-Editor oli ensimmäinen nimeämiskäytäntö. Pitkien keskustelujen pohjalta, eritoten Adele Goldbergin kanssa, päädyimme ilmaukseen Model-View-Controller." [Reenskaug, 2014]

Reenskaug suunnitteli MVC-mallin itse, mutta käytti sen ideoinnissa ja nimeämiskäytännöissä hyödyksi myös muiden tutkijoiden, kuten Adele Goldbergin, apua. Goldberg oli muun muassa *Smalltalk-80*-ohjelmointikielen oliokäsitteiden kehittämisessä mukana [Abbate, 2002].

Osmanin [2012b] mukaan MVC helpottaa sovelluksen toiminnallisuuksien modularisointia seuraavalla neljällä tavalla:

1) Ylläpito yksinkertaistuu, kun päivitysten yhteydessä kehittäjän on helppo päätellä, liittyykö muutos sovelluksen dataan, ohjelman toiminnan käsittelyyn vai käyttöliittymään.

2) Tiedon hakuun ja tallennukseen liittyvät yksikkötestit on helpompi toteuttaa, kun mallit ja näkymät on erotettu toisistaan selkeästi.

3) Matalan tason malli- ja ohjainkoodin toistoa saadaan vähennettyä.

4) Modulaarisuuden vuoksi sekä ohjelman ydinlogiikasta että käyttöliittymästä vastaavat kehittäjät voivat kehittää sovellusta yhtäaikaisesti.

Keskeinen tavoite MVC-mallin arkkitehtuurin toteuttavalla sovelluksella tulisi tämän pohjalta olla vastuualueiden selkeä erottelu, toistuvan koodin välttäminen ja yhtäaikaisen kehitystyön mahdollistaminen.

Tämän lisäksi tulisi MVC-mallin toteuttavien verkkopohjaisten sovellusten tarkastelussa erityisesti huomioida ohjaimen, näkymän ja varsinaisen käyttöliittymän välinen yhteys. Verkkosovellusten ohjain poikkeaa nimittäin toimenkuvaltaan työpöytäsovelluksissa käytetystä MVC-mallin ohjaimesta. Käyttöliittymätapahtumia ei voida verkkosovelluksissa suoraan liittää tiettyihin ohjaimen toimintoihin eli tapahtumankäsittelijöihin. DOM-elementti, johon tapahtuma liittyy, tunnistetaan kyseiseen DOM-elementtiin liitetyn *id*-tunnisteen tai *class*-luokkamäärittelyn perusteella, jolloin ohjain tulisi riippuvaiseksi käytetystä näkymästä, tai tarkemmin sanoen, näkymän käsittelemästä DOM-hierarkiasta (ks. listaus 1).

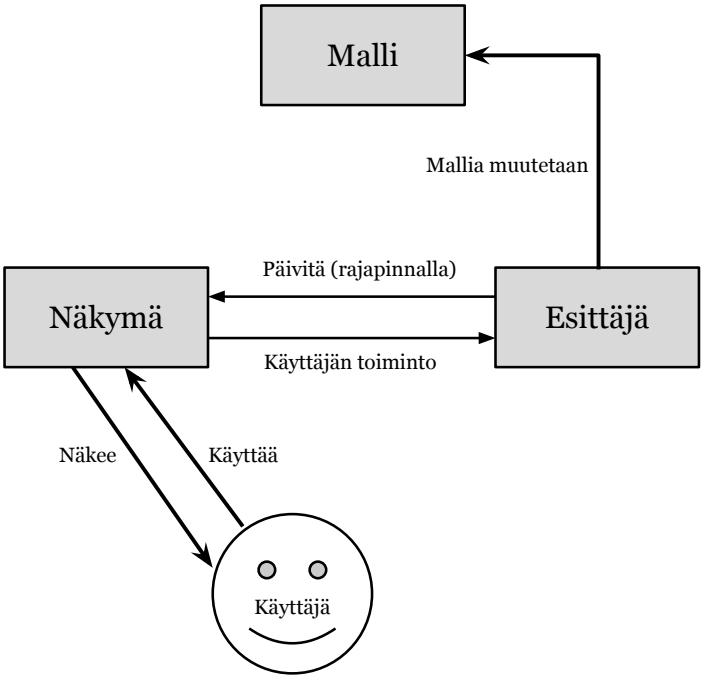
```
$('#okButton').click(function() {
    console.log('OK-nappia painettu.');
});
```

Listaus 1. Ohjaimen tapahtumankäsittelijä tulee näkymästä riippuvaiseksi, kun HTML-elementin nimeen viitataan suoraan.

Jotta ohjain saataisiin näkymästä riippumattomaksi, tulee liitos ohjaimen toimintojen ja näkymän takana olevan DOM-elementin välillä ilmaista joko näkymän puolella [Michaux, 2013] tai erillisessä tapahtumankäsittelijäluokassa. Joka tapauksessa on otettava huomioon myös sovelluksen MVC-arkkitehtuurin lopullinen toteutustapa. Esimerkiksi deklaratiivinen ohjelmointitapa saattaa oleellisesti vaikuttaa käyttöliittymätapahtumafunktioiden ja DOM-elementtien yhdistämiseen liittyvään toimintalogiikkaan.

2.3.2. MVP

MVP-mallin toteuttava sovellus koostuu mallista (engl. *model*), näkymästä (engl. *view*) ja esittäjästä (engl. *presenter*). MVP-mallissa perinteisesti ohjaimeen liittyviä tehtäviä on sijoitettu esittäjään, sillä näkymä on MVP-mallissa se sovelluksen osa, joka vastaanottaa käyttäjän syötteet. Näkymä ei kuitenkaan itse käsittele toimintoja, vaan pyytää esittäjää suorittamaan toimintoa vastaavan logiikan. Tähän voi sisältyä tiedon hakemista mallista tai mallin tiedon muuttamista. Pyydetyn toiminnon suorittamisen jälkeen esittäjä päivittää näkymän tilaa näkymän avaaman rajapinnan kautta, kuten kuvassa 2 on esitetty. [Osmani, 2012b]



Kuva 2. MVP-mallin näkymän, ohjaimen ja esittäjän vuorovaikutus suhteessa sovelluksen käyttäjään. [Manoj, 2012b]

Perinteisessä MVC-mallissa on muutama huono puoli: Ensinnä malli joutuu tiedottamaan näkymälle tilansa muuttumisesta. Toiseksi näkymän on oltava jossain määrin tietoinen mallista. Perinteisesti näkymän kuuluisi MVC-mallin mukaan toimia passiivisesti, "vain käskystä". MVP-mallissa näkymän ja mallin

välillä ei kuitenkaan tavallisesti ole erillistä rajapintaa, vaan esittäjä pyytää muokkaamaan mallin tilaa suoritettua toimintoa vastaavalla tavalla. Tällöin näkymästä saadaan passiivinen toimija. [Manoj, 2012a]

Keskeinen idea MVP-mallissa liittyy näkymän vastuualueen rajaamiseen. MVP-mallin ideana on pyrkiä ratkaisemaan kaksi keskeistä perinteisessä MVC-mallissa esiintyvää ongelmaa. Ensinnä malli joutuu tiedottamaan näkymille tilansa muuttumisesta. Toiseksi näkymä on täysin tietoinen mallissa. Näkymän tietoisuus mallista MVC-mallissa johtuu siitä, ettei näkymän ja mallin välillä ole erillistä rajapintaa. Tämän johdosta näkymä ei ole enää niin passiivinen kuin sen kuuluisi olla. [Manoj, 2012a]

MVP-mallin tarkoituksena on mallin ja näkymän välisen välittömän suhteen erottaminen. MVP-mallissa näkymä julkaisee sovitun mukaisen rajapinnan esittäjälle. Kun käyttäjä on vuorovaikutuksessa näkymän kanssa, näkymä kutsuu esittäjän metodia, jolloin esittäjä suorittaa vaaditun tehtävän, kuten tiedon haun mallista. Tämän jälkeen esittäjä päivittää jälleen näkymää käyttäen näkymän sovittua rajapintaa. [Manoj, 2012a]

MVP-mallin passiivisen näkymän tavoittelu saattaa monimutkaistaa sovelluksen käytännön toteutusta. Tämän vuoksi MVP-mallista on tehty kaksi eri versiota, joista toinen korostaa passiivista näkymää ja toinen aktiivisesti ohjaimen kaltaisena toimivaa näkymää. Näistä jälkimmäinen käyttää usein tietoliitoksia ja yksinkertaista koodia näkymän toimintojen toteuttamisessa. [Manoj, 2012a]

Alkuperäisen kuvan [Manoj, 2012b] pohjalta uudelleenpiirretty kuva esittää perinteisen, passiivista näkymää käyttävän MVP-mallin toimintalogiikan (kuva 2). Aktiivista näkymää käyttävän MVP-mallin toiminta on muilta osin sama, mutta lisäksi näkymä on vuorovaikutuksessa mallin kanssa tietoliitosten kautta. Keskeinen ero MVP- ja MVC-mallien välillä liittyy ohjaimen ja esittäjän erilaiseen tehtävään. MVP-mallissa esittäjän tehtävä on esittää näkymältä saadun kutsun mukaisen toiminnon tila näkymän kautta käyttäjälle [Manoj, 2012a]. MVC-mallissa ohjain ei toimi mallin ja näkymän välikappaleena, sillä ohjain ei päivitä näkymän tilaa. Sen sijaan ohjaimen tehtävänä on toimia välikappaleena käyttäjän aikaansaamien toimintojen ja mallin välillä, kun mallin vastuulle jää komentaa näkymää päivittymään mallin nykyisen tilan mukaiseksi.

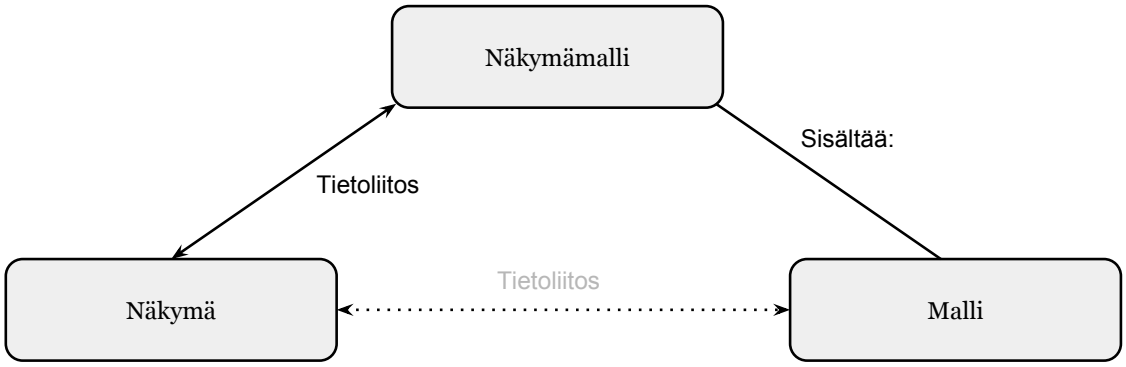
MVP-mallia saatetaan käyttää pääosin yritystason sovelluksissa silloin, kun sovelluksen esityslogiikan uudelleenkäyttö on keskeistä. MVC-mallin käyttö ei tällöin välttämättä tule kyseeseen, sillä usein monimutkaisten ja paljon käyttäjävuorovaikutusta sisältävien sovellusten toteuttamisessa näkymä joutuu

olemaan vuorovaikutuksessa useamman ohjaimen kanssa. MVP-mallissa taas kaikki sovelluksen monimutkainen logiikka voidaan eriyttää esittäjään, mikä voi merkittävästi helpottaa muun muassa ohjelman ylläpitoa. [Osmani, 2012b]

Esittäjä suorittaa sekä käyttöliittymätapahtumia vastaavan toiminnon että muokkaa käyttöliittymää toiminnon mukaisella tavalla. Täten MVP-pohjaisissa Javascript-sovelluskehyksissä sovelluksen näkymänä pidetään usein HTML-dokumentin DOM-rakennetta ja esittäjänä toimii olio, joka muokkaa käyttäjän syötteiden mukaisesti sovelluksen näkymän DOM-rakennetta.

2.3.3. MVVM

MVVM on MVC-malliin pohjautuva suunnittelumalli, jossa arkkitehtuuri jakautuu malliin (M), näkymään (V) ja näkymämalliin (VM). Suunnittelumallissa näkymä ja malli ovat liitoksissa toisiinsa näkymämallin välityksellä (kuva 3). Suunnittelumallissa myös näkymällä on keskeinen rooli sovelluksen toimintalogiikan kannalta: näkymässä määritellään tietoliitosten avulla tapahtumia, jotka muuttavat mallin tilaa.



Kuva 3. MVVM-mallissa näkymä on liitetty näkymämalliin tietoliitoksella, jolloin mallin data voidaan esikäsitellä näkymämallissa. [Mehran, 2010]

Malli pohjautuu MVC- ja MVP-malleihin. Mallin tarkoituksena on erottaa ohjelman toiminta ja toimintaan liittyvä data selkeästi sovelluksen käyttöliittymästä. Aluksi MVVM-malli suunniteltiin Windows-käyttöjärjestelmän uudeksi käyttöliittymien mallinnukseen käytettäväksi alijärjestelmäksi, mutta myöhemmin mallia on sovellettu Javascript-pohjaisten sovelluskehysratkaisujen arkkitehtuurissa. [Osmani, 2012c]

Mallin tehtävänä on – muiden MVC-suunnittelumallien tapaan – säilyttää ja käsitellä sovelluksen käyttötarkoitukseen liittyvää tietoa. MVVM-suunnittelumallin toteuttavassa sovelluksessa malli ei puutu sovelluksen toimintalogiikkaan eikä muotoile dataa millään tavoin. Ohjelman toimintalogiikka ja datan muotoilu kuuluvat MVVM-mallissa näkymämallin vastuualueeseen. [Osmani, 2012c]

MVC-mallin tapaan MVVM-mallissa näkymä on ainoa osa, johon sovelluksen käyttäjä on visuaalisesti yhteydessä. MVVM-mallissa näkymän tehtävänä on esittää näkymämallin nykyinen tila, ei suoraan mallin. Lisäksi MVVM-mallin näkymää pidetään aktiivisena toimijana, kun taas MVC-mallin näkymä on erillinen, passiivinen komponentti, joka ei itse ole tietoinen mallista. [Osmani, 2012c]

MVVM-mallissa näkymässä määritellään tietoliitosten avulla sovelluksen toiminta sekä käyttöliittymätapahtumat. Käyttöliittymätapahtumaan liittyvä funktio voidaan suorittaa näkymämallin puolella, mutta näkymä on silti vastuussa sovelluksen toimintalogiikan mukaisesta tapahtuman liittämisestä oikeaan näkymämallin määreeseen. [Osmani, 2012c]

Näkymämalli on vastuussa mallin tiedon muuntamisesta näkymässä esitettävään muotoon. Esimerkiksi tietokantamuodossa oleva päiväys voidaan muuntaa käyttäjän lokaalin mukaiseen muotoon. [Osmani, 2012c] Tässä mielessä näkymämalli on näkymän ja mallin välissä oleva muunnin-komponentti, joka vastaanottaa dataa mallilta ja antaa sen näkymän käyttöön. Lisäksi näkymämalli voi muuttaa mallin tilaa, näkymästä annettujen komentojen perusteella.

MVVM-mallissa näkymän tehtäväksi ei jätetä enää esitettävän tiedon muotoilua. Koska MVVM-mallisissa ohjelmissa mallista haettava data muotoillaan käyttäjälle näytettävään muotoon jo näkymämallissa, jää tältä osin näkymän tehtäväksi vain valmiiksi jalostetun tiedon esittäminen.

2.4. Suunnittelumallit

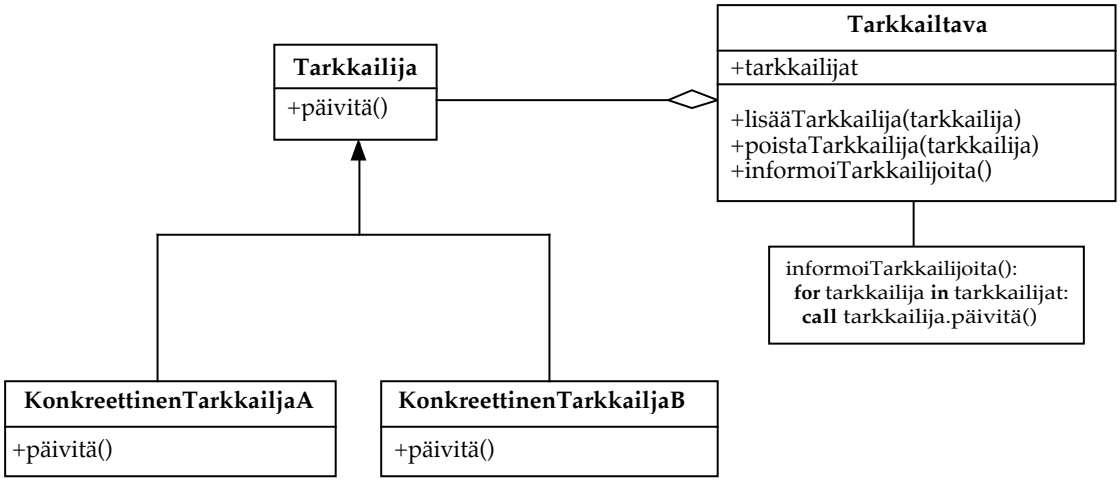
Suunnittelumalli on ohjelmistosuunnittelijoiden hyväksi havaitsema tapa ratkaista jokin ohjelmistokehitystyössä usein esiintyvä ongelma. Suunnittelu-malleja vastaavat toteutukset perustuvat nykyään yleensä oliopohjaisiin ratkaisuihin. Suunnittelumallien kuvaamiseen on kehitetty erilaisia valmiita kiinteitä malleja. Yleisimmässä tavassa esitellään suunnittelumallista neljä asiaa: ongelman nimi, itse ongelma, ongelman ratkaisu sekä siitä koituvat seuraukset. [Gamma et al., 1994]

Suunnittelumallit ovat keskeinen osa verkkosovelluskehyksiä. Sovellus-kehykset käyttävät usein useita suunnittelumalleja hyväkseen. Sovellus-kehyksen käyttö on helpompi oppia, sillä sovelluskehittäjät tuntevat yleisimmät suunnittelumallit.

Javascript-sovelluskehyksissä käytetään useita työpöytäsovelluksien kehittämiseen käytetyistä sovelluskehyksistä tuttuja suunnittelumalleja, kuten *tarkkailija-*, *komposiitti-*, *strategia-* *ja tehdasmetodimallia*. Näillä kaikilla on

keskeinen asema tasokkaan sovelluksen, ohjelmakoodin hallittavuuden ja yleisen arkkitehtuurillisen järjestyksen kannalta.

Tarkkailijamallissa olio sisältää luettelon oliota tarkkailevista olioista, jotta muutoksen sattuessa oliossa voidaan tarkkailevia olioita, niin kutsuttuja *tarkkailijoita* (kuva 4), huomauttaa muutoksen tapahtumisesta automaattisesti, ilman tarvetta tietää tarkkailijoiden tarkkaa toteutusta. Alkuperäisen kuvan [Observer, 2010] pohjalta piirretystä tarkkailijamallin kuvauksesta (kuva 4) nähdään, että olion, jossa muutos tapahtuu, ei tarvitse itse päivittää muiden olioiden tilaa. Sen sijaan kukin olio huolehtii itse, kuinka reagoi tarkkailtavassa oliossa tapahtuneeseen muutokseen. Näin ollen tarkkailijamallia käyttämällä saadaan osien välistä sidoksellisuutta vähennettyä. Tarkkailtavaa oliota kutsutaan paitsi *tarkkailtavaksi* myös *subjektiksi.* [Gamma et al., 1994]



Kuva 4. Tarkkailijamallin arkkitehtuuri. [Observer, 2010]

Pääasiallinen kommunikaatio MVC-mallin osien välillä tapahtuu tarkkailijamallin tai sen johdannaisen, kuten *Public-Subscribe*-mallin, toteutusta käyttäen. Siksi sitä pidetään tärkeimpänä MVC-mallin toteuttavan sovelluksen viestintätapana. Lisäksi tarkkailijamallin ominaisuudet mahdollistavat useamman mallin liittämisen yhteen näkymään MVC-mallissa. [Osmani, 2012b]

Komposiittimallissa tietorakenne voi koostua pienemmistä osista, jotka taas voivat koostua pienemmistä osista. Mallissa toimenpiteitä voidaan suorittaa kerralla kaikille mallia käyttäville aliolioille, kun mallin kaikki osat toteuttavat saman rajapinnan. [Gamma et al., 1994]

Strategiamallin avulla olion käyttäytymistä pystytään muuttamaan sovelluksen ajon aikana, jolloin olion toimintaa saadaan mukautettua itsenäisesti toisten olioiden käyttötarkoitusten mukaisesti. Strategiamallia käyttävä olio toteuttaa tietyn rajapinnan metodit, jotta oliota voidaan käyttää

tiettyyn käyttötarkoitukseen välittämättä olion yksityiskohtaisemmista metodimäärittelyistä. [Gamma et al., 1994]

Tehdasmetodi on malli, jossa abstraktin funktiokutsun kautta luodaan uusia olioita. Tämä on mahdollista, kun olion kutsutapa on määritelty, mutta itse luokan ilmentymän luominen tapahtuu kokonaan aliluokan sisällä. [Gamma et al., 1994]

Työpohjia (engl. *template*) käytetään muuttuvan näkymän sisällön runkona. Pohja määrittelee näkymän ulkoasun; muuttuville tiedoille on määritelty omat paikanvaraajansa, joihin kulloinkin esillä oleva data ilmennetään.

2.5. Oliomaailman käsitteet Javascriptissä

Javascript on *EcmaScript*-nimisen tulkattavan funktionaalisen kielen eräs toteutus. Sitä käytetään enimmälti verkkoselainsovelluksissa ja ylipäätään verkkosivuilla. Kieltä pidetään perinteisesti *tulkattavana* kielenä, vaikka jotkin ympäristöt kääntävät Javascript-koodin ennen ajamista.

Muita toteutuksia ovat muun muassa Microsoftin JScript [Microsoft, 2014] ja Google Chrome V8 [Google Developers, 2013]. Selkeyden vuoksi puhumme tässä tutkielmassa johdonmukaisesti Javascript-kielestä. Nimestään huolimatta kielellä ei ole yhteyttä Java-kieleen; nimi on valittu vain markkinointisyistä [Peltomäki & Nykänen, 2006].

Javascript-kielestä puuttuu kokonaan tuki *luokkien* käytölle; käytettävissä eivät ole perinteisistä ohjelmointikielistä tutut oliomaailman käsitteet, kuten *rakennin* (*engl. constructor)*, *purkumetodi* (*engl. destructor*), luokkamuuttujien näkyvyys, luokkien perintä ja rajapinnat. Näiden sijaan perinteisiä olioita vastaavat prototyyppipohjaiset ratkaisut; oliot ovat funktioita, joiden sisällä on muuttujia ja toisia funktioita. [Peltomäki & Nykänen, 2006]

Tästä voi koitua ongelmia, sillä suurin osa HTML-pohjaisista verkkosovelluksista toteutetaan juuri Javascriptiä käyttäen, vaikka yleisesti ohjelmistokehittäjät tuntevat luokkapohjaisen olio-ohjelmoinnin paradigman paremmin. Ratkaisuksi on kehitelty erilaisia Javascript-sovelluskehyksiä, joiden mukana tulee keinotekoinen tuki oliomaailmasta tutuille ratkaisuille. Nämä kehykset on toteutettu siten, että niiden käyttö ei vaadi lisäasennusta käyttäjän selaimeen, vaan kehys ladataan sivuston latauksen yhteydessä tavallisena Javascript-komentosarjana.

Seuraavaksi esittelen, kuinka Javascriptissä voidaan toteuttaa oliomaailmasta tutut käsitteet. Myös luokkapohjaisen oliomallin ja prototyyppipohjaisen ohjelmointitavan eroja käsitellään.

2.5.1. Näkyvyys

Olion jäsenmuuttujien ja -funktioiden *näkyvyyteen* voidaan Javascriptissä vaikuttaa tiettyyn mittaan asti samantapaisesti kuin perinteisissä olio-pohjaisissa kielissä. Näkyvyydessä on kyse luokan sisällä olevan jäsenen käytettävyydestä luokan ulkopuolelta käsin [Loudon, 2010]: Paikallista eli suojattua metodia voi käyttää vain luokan sisältä. Julkista jäsentä voi käyttää luokan ulkopuoleltakin. Näkyvyydestä puhuttaessa viitataan myös käsitteisiin *tiedon kätkentä* [Loudon, 2010] ja luokan tietojen *kapselointi* [Sommerville, 2010].

Kun muuttuja tai funktio liitetään olioon *this*-viitteen avulla, metodista tai muuttujasta tulee julkinen, eli siihen pääsee käsiksi myös olion ulkopuolelta (ks. listaus 2).

```
function LuokkaA() {
  this.nimi = "Jouni";
  this.teeJotain = function() {};
}
var a = new LuokkaA();
console.log(a.nimi); // Tuloksena "Jouni".
```

Listaus 2. Julkisen jäsenmuuttujan käyttö olion ulkopuolelta.

Kun muuttuja tai funktio liitetään olion sisällä käyttäen *var*-määrettä, funktioon tai muuttujaan pääsee käsiksi vain olion sisältä (ks. listaus 3). Toisaalta tällaisiin, paikallisiin muuttujiin pääsee käsiksi olion sisäisistä metodeista, kuten esimerkin saantimetodista *annaNimi()*.

```
function LuokkaB() {
  var nimi = "Jouni";
  var teeJotain = function() {};
  this.annaNimi = function() {
    return nimi;
  }
}
var b = new LuokkaB();
console.log(b.nimi); // Tuloksena undefined, määrittelemätön.
console.log(b.annaNimi()); // Tuloksena "Jouni".
```

Listaus 3. Paikallinen jäsenmuuttuja ja -funktio.

Sen sijaan suojattuja metodeja (engl. *protected methods)* ei voida perinteisin keinoin toteuttaa Javascriptissä. Suojatut metodit ovat käytettävissä luokan lisäksi luokan perineistä luokista käsin [Loudon, 2010]. On olemassa kuitenkin sovelluskehyksiä, jotka lisäävät tuen suojattujen metodien käytölle.

2.5.2. Rakennin

Rakennin on luokan funktio, jota kutsutaan, kun luokasta luodaan ilmentymä, olio [Loudon, 2010]. Javascript ei tunne perinteistä *rakennin*-käsitettä. Sen sijaan, kun oliosta luodaan ilmentymä *new*-operaattorilla, sen rungossa määritellyt käskyt suoritetaan niiden määrittelyjärjestyksessä. Tässä mielessä rakennin on koko oliofunktio. [Peltomäki & Nykänen, 2006]

Tästä seuraa, että erillinen rakenninmetodi on mahdollista luoda keinotekoisesti: jäsenmuuttujien määrittelemisen jälkeen, luokan lopussa, voidaan kutsua esimerkiksi olion omaa *init*()-metodia, jolloin metodi tulee automaattisesti kutsutuksi aina, kun oliosta luodaan uusi ilmentymä (ks. listaus 4).

```
function Henkilo() {
  this.ika = null;
  this.nimi = null;

  this.init = function() {
    this.ika = 27;
    this.nimi = "Jouni";
  }

  this.init();
}

var henkilo = new Henkilo(); //Tässä kutsutaan myös init()-metodia.
```

Listaus 4. Javascriptin rakentimen käytännön toteutus.

2.5.3. Perintä

Perinnän avulla luokka saa perityn luokan ominaisuudet. Perittyjä luokkia kutsutaan *aliluokiksi*. Javascriptissä perintä saadaan aikaan asettamalla funktio-olion prototyypiksi haluttu muu olio. Jokaisella oliolla on sisäinen *prototype*-niminen jäsenmuuttuja, joka voidaan asettaa viittaamaan toiseen olioon. Olio saa käyttöönsä kaikki viitatun prototyypin jäsenmuuttujat ja -funktiot, ja niitä voi kutsua olion ulkopuolelta aivan kuin ne olisivat olion omia muuttujia ja funktioita (ks. listaus 5).

```
function Henkilo(nimi, syntymavuosi) {
  this.nimi = nimi;
  this.syntymavuosi = syntymavuosi;

  this.ika = function() {
    var kuluvaVuosi = new Date().getFullYear();
    return kuluvaVuosi - this.syntymavuosi;
  }
}

function Opiskelija(nimi, syntymavuosi, opiskelijanumero) {
  this.nimi = nimi;
  this.syntymavuosi = syntymavuosi;
  this.opiskelijanumero = opiskelijanumero;
  this.valmistunut = false;
}

Opiskelija.prototype = new Henkilo();
var matti = new Opiskelija("Matti", 1917, 12345);
console.log( matti.ika() ); //Tulos: 2014-1917
```

Listaus 5. Prototyyppipohjainen perinnän toteutus.

Tässä on tärkeää huomata, että Javascriptissä ei ole mahdollista luoda muulta ympäristöltä suojattua tietorakennetta eli luokkaa, vaan luokkien sijaan luodaan yleiskäyttöisiä olioita [Loudon, 2010]. Tämä tarkoittaa sitä, että Javascriptissä *new*-operaattorilla ilmennetään oliosta toinen olio, ilmentymä, ei niin, että luokasta ilmennetään olio, kuten luokkapohjaisissa kielissä on käytäntönä.

Olion prototyypin asettamisella käyttäen olion ilmentymää on kuitenkin omat haittansa. Koska ilmentymä luodaan *new*-operaattoria käyttäen, tulee prototyypin asettamisen yhteydessä kutsuttua samalla perittävän olion rakenninta, mikä ei ole tarkoituksenmukaista. Lisäksi joudutaan joka tapauksessa toistamaan perityn olion rakentimen koodi perivän olion rakentimessa, mikä on vastoin ohjelmistokehityksen yleistä *DRY*-periaatetta *(Don't repeat yourself)*, jonka mukaan toistuvaa koodia tulisi välttää.

Sen sijaan perittävän olion rakenninta tulisi kutsua vasta perivän olion rakentimesta, jotta perittävän olion rakentimessa asetetut jäsenmuuttuja-arvot vaikuttavat myös perivän olion arvoihin. Tämä vastaa luokkapohjaisissa kielissä yläluokan rakentimen kutsumista alaluokan rakentimesta käsin. Tämä ongelma on ratkaistu Javascriptin uudemmassa standardissa, *EcmaScript 5*:ssä, *Object.create()*-nimisellä metodilla [ECMA-262, 2011]. Sen avulla oliosta voidaan luoda ilmentymä rakenninta kutsumatta. Näin perittävän olion rakenninta voidaan prototyypin asettamiseen liittyvän koodin sijaan kutsua vasta perivän

olion rakentimesta käsin, käyttäen Javascriptin sisäänrakennettua *call()*-metodia [Mozilla, 2014b], kuten listaus 6 osoittaa.

```
function Henkilo(nimi, syntymavuosi) {
  this.nimi = nimi;
  this.syntymavuosi = syntymavuosi;

  this.ika = function() {
    var kuluvaVuosi = new Date().getFullYear();
    return kuluvaVuosi - this.syntymavuosi;
  }
}

function Opiskelija(nimi, syntymavuosi, opiskelijanumero) {
  Henkilo.call(this, nimi, syntymavuosi);
  this.opiskelijanumero = opiskelijanumero;
  this.valmistunut = false;
}
Opiskelija.prototype = Object.create(Henkilo);
```

Listaus 6. Perinnän toteutus huomioiden perityn olion rakentimen kutsuminen.

Perinnän toteuttamiseen Javascript-kielellä on kehitelty myös useita erilaisia kolmannen osapuolen ratkaisuja (ks. esimerkiksi Classify.js [2010]). Nämä mahdollistavat muiden olio-ominaisuuksien rinnalla perintätuen, joka muistuttaa enemmän perinteisiä ohjelmointikieliä. Esimerkiksi moniperintä on osassa ratkaisuja mahdollista (ks. esimerkiksi Ring.js [2013]).

On olemassa sekä kirjastoja, jotka keskittyvät ainoastaan perinteisistä kielistä tuttujen luokkatoiminnallisuuksien, kuten perinnän, tuomiseen Javascript-kieleen, että sellaisia Javascript-sovelluskehyksiä, joissa perintää sekä muita oliopohjaisia ratkaisuja tuetaan niin ikään sovelluskehyksen omien toiminnallisuuksien ohella.

3. Sovelluskehykset

Tässä luvussa perehdytään sovelluskehyksen ideaan ja tilanteisiin, joissa sovelluskehyksen käytöstä on hyötyä. Esiin tuodaan myös erilaisia sovelluskehystyyppejä. Lopuksi esitellään vertailussa myöhemmin käytettävät laatuominaisuudet.

3.1. Yleistä sovelluskehyksistä

Sovelluskehyksessä on kyse valmiista ohjelmarungosta tai sen osasta, johon puuttuvat kohdat täydentämällä saadaan aikaan toimiva sovellus tai sen osa [Haikala & Märijärvi, 2002]. Näin ollen sovelluskehys koostuu abstrakteista ja konkreettisista luokista ja niiden välisistä rajapinnoista [Wirfs-Brock & Johnson, 1990].

Usein sovelluskehyspohjaiset sovellukset rakennetaan lisäämällä sovellukseen komponentteja ja luomalla sovelluskehyksen abstrakteista luokista sovelluskohtaisia toteutuksia. Sovelluskehys ei itsessään ole sovellusohjelma, vaan sovellusohjelma koostetaan usein useaa eri sovelluskehystä käyttäen. [Sommerville, 2010]

Sovelluskehyksen abstraktien luokkien toteuttamisen lisäksi sovelluskehystä käyttävässä sovelluksessa on usein määritettävä *takaisinkutsufunktioita* (engl. *callback function*). Sovelluskehys kutsuu näitä funktioita tiettyjen sovelluskehyksen havaitsemien tapahtumien yhteydessä. [Sommerville, 2010] Tämä on osa sovelluskehysten laajennettavuutta tukevaa toiminnallisuutta.

Sovelluskehysten lähtökohtana on usein joukko erilaisia suunnittelumalleja. Käyttöliittymäpainotteisissa sovelluskehyksissä, kuten nimenomaan verkkosovelluskehyksissä, nousee keskeisenä suunnittelumallina esiin MVC-malli sekä muut sen yhteydessä käytettävät oheismallit, kuten tarkkailijamalli. [Sommerville, 2010]

3.2. Sovelluskehyksen käytön etuja

Selainpohjaisen sovelluksen kehittäjän pitää osata hyödyntää lukuisia erilaisia tekniikoita sovelluksen kehittämisessä. Verkkosovelluksen kehittäjän on osattava käyttää palvelinpuolen ohjelmistotekniikkojen lisäksi HTML-merkkauskieltä, määritellä sivuston tyylejä ja asettelua tyylimerkkauskielillä, käyttää mahdollisesti useita graafisia työkaluja sekä hallita Javascript-kielen käyttö.

Koko verkkosovelluksen toteuttaminen suoraan "puhtaalla" Javascriptillä koodaten saattaa osoittautua mahdottomaksi ratkaisuksi useista syistä. Suoraan

koodaten sovelluksen kehitystyö voi hidastua, kun jokainen asiakokonaisuus pitää kirjoittaa koodiin yksityiskohtaisesti ja pitkäsanaisesti. Myös kehittäjien riittämätön tietotaito eksplisiittisestä sovelluksen toteutuksesta Javascriptillä saattaa johtaa kehitystyön hidastumiseen. Kehitystyön hidastumisen lisäksi koodin laatu voi kärsiä, jos kehittäjä ei kiinnitä sovelluksen rakenteelliseen toteutukseen riittävästi huomiota.

Vastakohtana suoraan Javascriptillä koodaamiseen on sovelluskehyksen käyttäminen. Sovelluskehykset helpottavat kehittäjän taakkaa, kun ohjelman tekijä saa keskittyä enemmän sovelluksen logiikan rakentamiseen kuin pienten, usein toistuvien ohjelmakokonaisuuksien, kuten käyttöliittymäkomponentti-rajapintojen, yksityiskohtien toteuttamiseen. Sovelluskehykset asettavat sovelluksen kehittämiseen rakenteelliset rajat sekä tarjoavat usein käytettäviin toimintoihin valmiita ratkaisumalleja. Näin kehittäjä itse sekä mahdolliset muut kehittäjät hyötyvät siitä, että koodin selkeän rakenteen myötä sovelluksen rakenneosat ovat löydettävissä odotettavista paikoista.

Lisäksi sovelluskehykset tarjoavat valmiita funktioita ja käyttöliittymäkomponenttien käsittelyyn ja paikantamiseen tarvittavia navigointisääntöjä. Sovelluskehystä käyttäessään kehittäjän ei tarvitse olla tietoinen eksplisiittisestä Javascript-toteutuksesta itsessään, sillä yksityiskohtien toteutus sekä selainten välisten toimintaerojen huomioonottaminen sisältyy jo sovelluskehyksen toimintamalliin.

Lisäksi sovelluskehyspohjaisten sovellusten uudelleenkäyttö ja mukautus eri käyttökohteisiin on yksinkertaisempaa, sillä samoja toiminnallisuuksia ei tarvitse tehdä joka projektia varten uudestaan. Koska sovelluskehyksen toteutuksen koodi on eriytetty sovellustoteutusten koodista, voidaan sovelluskehystä päivittää sovelluskehystä käyttävien sovellusten toteutustavasta välittämättä.

3.3. Erilaiset sovelluskehysluokat

Sovelluskehyksiä on kehitetty tietotekniikan historian aikana hyvin laajasti eri soveltamisaloihin. Yleensä sovelluskehyksen valinnassa on otettava huomioon sovelluksen toteutustapa, käyttötarkoitus sekä mahdollisesti yrityksen omat toimintaohjeet.

Yleisellä tasolla sovelluskehykset voidaan jakaa kolmeen luokkaan [Fayad & Schmidt, 1997]:

1) *Järjestelmän infrastruktuuriin liittyvät kehykset* tukevat järjestelmän infrastruktuurin, kuten tiedonvälityksen, käyttöliittymien ja kääntäjien käyttöä.

2) *Väliohjelmiston yhdentämiseen liittyvät kehykset* koostuvat joukosta standardeja ja niihin liittyviä olioluokkia, joiden avulla komponenttien välinen viestintä helpottuu.

3) *Yritysten sovelluskehykset* liittyvät erikoisiin sovellusaloihin, kuten televiestintään tai rahajärjestelmiin, jotka sulauttavat sovellusalojen tietämystä ja tukevat loppukäyttäjien sovellusten kehittämistä.

Verkkosovellusten kehittämiseen käytetyt kehykset painottuvat pitkälti käyttöliittymäpohjaisten sovellusten tukemiseen. Jaan verkkosovellusten kehittämiseen tarkoitetut kehykset viiteen eri luokkaan (ks. taulukko 1). Jaottelun tarkoituksena on auttaa hahmottamaan laajempaa kuvaa verkkosovelluskehysten maailmasta. Taulukon 1 luokat eivät välttämättä ole toisiaan poissulkevia, vaan niitä voi osittain käyttää samassa projektissa yhtä aikaa, eri käyttötarkoituksiin. Lisäksi jaottelu ei ole täysin yksikäsitteinen; tällä tarkoitan sitä, että yksi kehys voi tiettyjen toiminnallisuuksien vallitessa kuulua useampaankin eri luokkaan.

Kehystyyppi	**Kehysesimerkkejä**
1. Yleiskäyttöisiä kehyksiä tai kirjastoja	jQuery, Mootools, prototypejs
2. Käyttöliittymäkehyksiä	AngularJS, Spine, Sproutcore, Ext.js, jQuery UI, Dojo Toolkit
3. Javascriptin http-palvelin-kehyksiä	Node.js
4. MVC-pohjaisia Javascript-kehyksiä	Maria, Backbone.js, JavaScriptMVC (jQueryMX), PureMVC
5. Käännettäviä Javascript-kehyksiä (Ohjelma kehitetään muulla kielellä, mutta selainta varten ohjelma käännetään Javascript-muotoon.)	Google Web Kit (pohjana Java), Forthkit (kielenä ObjectiveC), Vaadin Framework (kielenä Java), Cappuccino Framework (kielenä Objective-J, joka laajentaa Javascriptiä Objective-C-syntaksilla)

Taulukko 1. Verkkosovelluskehystyypit käyttötarkoitusten mukaan luokiteltuna.

Yleiskäyttöiset kehykset tarjoavat valmiita ratkaisuja HTML-puun DOM-rakenteen läpikäyntiin, muokkaamiseen ja käyttämiseen. Ne helpottavat myös usein käytettyjä Javascript-toimintatapoja tarjoamalla yksinkertaisemman tavan yleisten toimintojen lopputuloksen aikaansaamiseen. Osa myös helpottaa oliomaailman toimintatapojen suoraviivaista käyttöä Javascript-ympäristössä.

Käyttöliittymäkehykset tarjoavat valmiita ratkaisuja HTML:n omien komponenttien käytölle. Ne myös tuovat mukanaan sellaisia käyttöliittymäkomponentteja, joita ei perustason HTML-merkkauskielen mukana tule. Tämän tyypin kehykset myös tarjoavat rajapinnat komponenttien muuntamista, käyttöliittymätapahtumien käsittelyä ja komponenttien teemoitusta varten.

Http-palvelinkehykset mahdollistavat rajoitetun http-palvelimen käynnistämisen ja ajamisen asiakkaan puolella, jolloin varsinainen palvelin toimii tietoliikenneyhteyksien ja http-pyyntöjen käynnistäjänä ja selaimen osa toimii palvelimena. Näin voidaan toteuttaa vuorovaikutteisia sovelluksia, kun käyttäjän selain ei joudu kyselemään (engl. *poll*) tietyin väliajoin palvelimelta, onko uutta dataa (esimerkiksi keskusteluohjelmassa uusi viesti) saatavilla palvelimelta.

Käännettävät Javascript-kehykset ovat siitä erikoisia, että niiden avulla kehittäjä voi käyttää jotain muuta kieltä selainpohjaisen koodin toteuttamiseen. Tällöin kirjoitettu koodi joko esikäännetään Javascript-muotoon ja selain käyttää Javascriptille valmiiksi käännettyä koodia sivuston toiminnallisuuksien toteuttamisessa, tai koodi käännetään "lennossa" Javascript-muotoon sivun lataamisen yhteydessä. Jälkimmäistä käytetään varsinkin sovelluksen kehitysvaiheessa, jolloin kääntämättömään koodiin tehdään muutoksia lähes jokaisella sivunlatauskerralla.

MVC-pohjaisten Javascript-kehysten mukana ei tule välttämättä käyttöliittymäkomponentteja, vaan niiden tehtävänä on yksinomaan määritellä puitteet sovelluksen toteuttamiseen MVC-arkkitehtuuria käyttäen, jättäen kehittäjälle vapaat kädet haluamansa käyttöliittymäkirjaston valitsemiseen tai valitsematta jättämiseen.

3.4. Laatuominaisuusstandardit

Sovelluksen arkkitehtuurin, käytettävyyden ja muiden ominaisuuksien mittaamiseen voidaan käyttää tiettyjen standardien mukaisia laatuominaisuuksia ja niiden esiintyvyyttä sovelluksessa. Standardeja ovat kehittäneet muun muassa ISO-, ANSI- ja IEEE-standardointijärjestöt.

Yleisimmät ohjelmiston laadun määrittelyyn käytetyt tavat ovat [Jørgensen, 1999]:

1) Ohjelmiston laatu päätellään jonkin standardin, kuten ISO 8402-1986 tai IEEE 610.12-1990, laatutekijöiden perusteella.

2) Ohjelmiston laatu päätellään käyttäjätyytyväisyyden pohjalta.

3) Ohjelmiston laatu päätellään sen mukaan, kuinka paljon ohjelmistossa on virheitä tai odottamattomia toimintoja.

Ohjelmiston laadun mittausstandardit perustuvat pitkälle intuitioon laadun olemassaolosta. Esimerkiksi käyttäjätyytyväisyyteen perustuva laatustandardi pohjautuu mielipiteeseen tietystä tavoitettujen käyttäjätarpeiden tasosta. [Jørgensen, 1999]

3.4.1. ISO-8402

ISO-8402 määrittelee laatuun liittyviä termejä ja selventää standardoinnin avulla muun muassa laadunhallintaan ja -tarkkailuun liittyviä käsitteitä. Standardi määrittelee ohjelmiston laadun "kokonaisuutena ominaisuuksia tai ominaispiirteitä jossakin tuotteessa tai palvelussa, joihin sen kyky tavoittaa tuotteeseen kohdistuvat odotukset perustuu". Seuraavat kuusi ominaisuutta kuuluvat tällaisiin standardin määrittelemiin ominaispiirteisiin eli laatutekijöihin [ISO-8402, 1994]:

- tehokkuus
- joustavuus
- yhtenäisyys
- yhteentoimivuus
- ylläpidettävyys
- siirrettävyys.

Tehokkuus kuvaa tuotteen, tässä tapauksessa tietokoneohjelman, kykyä täyttää sen tarkoituksenmukainen toiminta riittävällä resurssimäärällä. *Joustavuus* mittaa, kuinka helposti ohjelmaan voidaan lisätä toiminnallisuutta. Jos ohjelman rakenne on suunniteltu laajennettavuutta edistävien toimintamallien mukaisesti, ohjelman toiminnan sovittaminen asiakkaan uusien vaatimusten mukaisesti on helpompaa. *Yhtenäisyys* (engl. *integrity*) eli eheys kuvaa ohjelman kykyä toimia sille asetettujen toimintaodotusten mukaisesti. *Yhteentoimivuus* (engl. *interoperability*) kuvaa ohjelman kykyä toimia yhdessä jonkin toisen ohjelman kanssa. *Ylläpidettävyys* mittaa, kuinka paljon työtä vaaditaan ohjelman vikojen ja muutosvaatimusten täyttämiseen. *Siirrettävyys* kuvaa, kuinka paljon työtä tarvitaan ohjelman siirtämisessä ympäristöstä toiseen. [ISO-8402, 1994; Ronan, 1996]

Standardia voidaan soveltaa niin tuotteiden kuin palvelujen laadun varmistukseen. Palvelun tai tuotteen laatu ei kuitenkaan koskaan koostu vain luettelosta eri laatuominaisuuksia. Radziwillin [2008] mukaan standardin määritelmässä *kokonaisuus* viittaa siihen, että laatuominaisuuksien lisäksi laadussa on kyse myös sovelluksen "suunnittelusta, toteutuksesta sekä näiden ominaisuuksien ja yksilön välisestä vuorovaikutuksesta". Näin ollen voitaneen päätellä, että sovelluksen laadun arvioinnissa tulisi laatuominaisuuksien

erillisen tarkastelun lisäksi kiinnittää huomiota ominaisuuksien väliseen vuorovaikutukseen sekä sovelluksen arkkitehtuurin tarkasteluun myös yleisellä tasolla.

ISO-8402 on nykyään vanhentunut standardi, joten sen tarkempi tarkastelu ei liene mielekästä. Standardin ovat korvanneet uudemmat laadun määrittelystandardit, kuten ISO-9126.

3.4.2. ISO-9126

ISO-9126-standardia käytetään ohjelmistojen sisäisen ja ulkoisen laadun mittaamiseen. Tällä hetkellä se kuuluu yleisimpiin ohjelmistolaadun mittaamiseen käytettäviin standardeihin. Standardilla on ohjelmistotuotannon kannalta keskeinen merkitys, sillä se perustuu nimenomaan ohjelmistojen laadun mittaamiseen. Nykyisessä muodossaan se sisältää sekä laatuun liittyviä malleja että laadun mittaamiseen käytettäviä mittareita. [Botella et al., 2004]

Standardi jakautuu neljään osaan: *laatumalli, sisäiset mittaukset, ulkoiset mittaukset* ja *käytön laatu*. Sisäisillä mittauksilla mitataan sellaisia toiminta-kokonaisuuksia, jotka eivät liity ohjelman suorittamiseen. Ulkoisia mittaus-menetelmiä taas voidaan soveltaa suoritettavaan ohjelmaan. Käytön laatua voidaan standardin mukaan mitata, kun ohjelma on valmis ja käytössä. [ISO-9126, 2000]

Tässä standardin tarkastelu rajataan laatumallin esittelyyn. Laatumalli koostuu seuraavista laatuattribuuteista [ISO-9126, 2000]:

- toiminnallisuus
- luotettavuus
- käytettävyys
- tehokkuus
- ylläpidettävyys
- siirrettävyys.

Kukin näistä attribuuteista jakautuu vielä useampaan aliattribuuttiin. Esimerkiksi toiminnallisuus jakautuu seuraaviin aliattribuutteihin: Soveltu-vuus, tarkkuus, yhteentoimivuus ja turvallisuustoiminnan noudattaminen. [ISO-9126, 2000]

Standardin mukaan *toiminnallisuus* kuvataan joukolla attribuutteja, jotka koostuvat joukosta funktioita ja määrättyjä funktiosidonnaisia ominaisuuksia. *Luotettavuus* eli vakaus koostuu joukosta attribuutteja, jotka aiheutuvat ohjelmiston kyvykkyydestä säilyttää suorituskykynsä ja tehokkuutensa määrätyissä olosuhteissa määrätyn ajan.

Käytettävyys koostuu attribuuteista, jotka aiheutuvat siitä, kuinka paljon käyttäjä joutuu ponnistelemaan voidakseen käyttää ohjelmaa ohjelman käyttötarkoituksen mukaisesti. Käytettävyyden osalta ominaisuuden arvioijina toimivat sovelluksen todelliset tai oletetut käyttäjät. *Tehokkuutta* kuvataan standardissa joukolla määritteitä, jotka vastaavat ohjelman suorituskykytason ja ohjelmaan käytetyn resurssimäärän välillä olevaa suhdetta. *Ylläpidettävyys* koostuu joukosta määritteitä, joihin vaikuttaa määrättyjen muutosten tekemiseen tarvittava työpanos. *Siirrettävyydellä* taas kuvataan sovelluksen kykyä siirtyä ympäristöstä toiseen. [ISO-9126, 2000]

Standardissa määritellään attribuuttien ja aliattribuuttien lisäksi vielä lisäattribuutteja kunkin aliattribuutin alla. Lisäattribuutti on kokonaisuus, joka voidaan varmistaa tai mitata ohjelmistotuotteesta. Lisäattribuutteja standardi ei määrittele, sillä ne vaihtelevat eri tuotteiden välillä. [ISO-9126, 2000]

3.4.3. Keskeiset laatuattribuutit

Arvioinnin tueksi otan ISO-9126-standardin joukosta kolme keskeistä laatuattribuuttia tarkastelun kohteeksi: *tehokkuus*, *ylläpidettävyys* ja *siirrettävyys*. Nämä kolme attribuuttia esiintyvät myös ISO-8402-standardin määrittelyssä. Attribuuteista *toiminnallisuus*, *luotettavuus* ja *käytettävyys* jäävät pois arvioinnin tukena käytettävistä ominaisuuksista. Mukaan otettavien kolmen attribuutin lisäksi arvioinnissa otetaan huomioon verkkosovelluskehyspohjaisten sovellusten arkkitehtuurin arvioinnissa oleellisiksi kokemani neljä muuta laatuattribuuttia: *sidoksellisuus*, *luettavuus*, *laajennettavuus* ja *koko*.

Kehysten pohjalta toteutettavien sovellusten arkkitehtuuria tarkasteltaessa en koe otolliseksi ottaa tarkasteluun mukaan luotettavuuden käsitettä. Luotettavuus ei välttämättä sovi laatuominaisuutena tämän tutkielman kehyksien vertailuun, sillä kyseessä ei tietyssä mielessä ole kehyskohtainen laatuominaisuus: lähtökohtaisesti kehykset ovat toiminnallisuuksiltaan vakaita ja tietoturvaltaan taattuja. Luotettavuuden näkökulma sopisi tarkasteltaviin laatuominaisuuksiin mukaan paremmin, mikäli vertailtaisiin sovellusohjelmia yleisesti. Nyt on kyse sovelluskehysten pohjalta tehtävien sovellusten laatuominaisuuksista, joten luotettavuuden kannalta oleellista olisi tarkastella vain toteutuskohtaisista ratkaisuista aiheutuvia sovelluksen luotettavuutta, toimintavarmuutta ja tietoturvaa heikentäviä tai vahvistavia tekijöitä.

Lisäksi käytettävyyden käsite rajautuu pois tutkielmassa tarkasteltavista laatuominaisuuksista. Käytettävyyteen vaikuttavat yleensä sovelluskohtaiset käytännön ratkaisut, joihin ei ole käytetyllä sovelluskehyksellä merkittävää vaikutusta. Vastavuoroisesti käyttöliittymäkomponentteja tarjoavissa verkko-

pohjaisissa sovelluskehyksissä olisi käytettävyyden käsitteellä keskeinen osa arvioinnin ja vertailun toteuttamisessa.

Lopulta ISO-standardin *toiminnallisuuden* käsitettä ei myöskään ole sisällytetty vertailuun. Se on käsitteenä hyvin moniselitteinen ominaisuus. Lisäksi kehysten toiminnallisuutta tarkastellaan jo kehysten esittelyn – ja osittain kehysten arvioinnin – yhteydessä.

Sen sijaan standardien pohjalta valittujen laatuominaisuuksien lisäksi ovat neljä muuta laatuominaisuutta – *sidoksellisuus, luettavuus, laajennettavuus* ja *koko* – keskeisiä vertailtaessa verkkosovelluskehyksiä. Sidoksellisuuden tarkastelu saattaa sovelluskehysten avulla toteutettavien sovellusohjelmien arkkitehtuuria tarkasteltaessa osoittautua tärkeäksi, sillä kehysten arkkitehtuurin pohjalla oleva MVC-malli pyrkii sovelluksen osien välisen sidoksellisuuden vähentämiseen. Siksi on ehkä tärkeää tutkia, miten sidoksellisuutta on eri kehyksissä saatu vähennettyä ja millaisia eroja sidoksellisuuden esiintyvyydessä on havaittavissa.

Luettavuuden tarkastelun lähtökohtana on eri kehyksillä tehtyjen sovellusten ohjelmakoodin ymmärrettävyys. Tämä on tarkastelussa mukana, sillä lähtöoletuksena luettavuuden luulisi vaikuttavan päivitettävyyden kautta sovelluksen ylläpidettävyyteen sekä kanssakäymisen mahdollistumiseen eri ohjelmistokehittäjien välillä. Laajennettavuuden tarkastelu taas on sidoksissa niin ikään sovelluksen ylläpidettävyyteen, sillä nykyaikaisten sovellusten käyttötarpeet todennäköisesti muuntuvat asiakkaan vaikutuksesta muuttuneiden olosuhteiden myötä ainakin jollain tasolla koko käyttöajan aikana. Lopulta verkkopohjaisilla sovelluksilla voisi olettaa myös sovelluskehyksen ja tehtävän koodin koolla olevan merkitystä, sillä sovellukset ladataan käyttäjän selaimeen verkkoyhteyden välityksellä.

3.5. Laatuominaisuudet

Ohjelmistotekninen viitekehys määrittää hyvin pitkälle, mitkä laatuominaisuudet ovat tarkasteltavien sovellusten kannalta keskeisiä. Tutkielmassa tarkastellaan verkkopohjaisten sovelluskehysten avulla toteutettavien sovellusten arkkitehtuuria, joten laatuominaisuuksia tarkasteltaessa on syytä keskittyä kehysten avulla toteutettavan sovelluksen ohjelmistoteknisistä ratkaisuista ilmenevien laatuominaisuuksien esiintyvyyteen. Tarkastelen seuraavaksi yleisellä tasolla seitsemää eri laatuominaisuutta:

1) sidoksellisuus
2) luettavuus
3) tehokkuus
4) siirrettävyys
5) ylläpidettävyys
6) laajennettavuus
7) koko.

Tuon laatuominaisuuksista lyhyesti esiin myös keskeisimpiä huomioita niiden merkityksestä verkkosovellusten kehityksen näkökulmasta tarkasteltuna.

3.5.1. Sidoksellisuus

Sidoksellisuudessa (engl. *coupling*) on kyse ohjelmamoduulin riippuvuudesta ohjelman toisista osista. Ohjelmassa voi esiintyä *tiukkaa sidoksellisuutta* (engl. *tight coupling*) tai *löyhää sidoksellisuutta* (engl. *loose coupling*). Käytännön tasolla sidoksellisuuteen vaikuttaa se, missä määrin luokka on suoraan tietoinen toisen luokan toteutuksesta. [Yoon, 2014]

Sovelluksen tai sen osan sidoksellisuudessa ei ole pelkästään kysymys luokan kapseloinnista tai sen puuttumisesta. Kyse ei myöskään ole siitä, kuinka tarkasti toinen luokka tietää toisen luokan attribuutit tai toteutuksen. Kun luokka on *suoraan* tietoinen toisen luokan määritelmästä, esiintyy tiukkaa sidoksellisuutta. Käytännön toteutuksessa sidoksellisuutta ilmenee silloin, kun luokka sisältää suoran viittauksen toiseen vaaditun toiminnallisuuden sisältävään konkreettiseen luokkaan. Tässä mielessä tiukassa sidoksellisuudessa on kyse luokan riippuvuudesta toisen luokan toteutuksesta. [Kaye, 2003]

Löyhä sidoksellisuus on usein tavoiteltava tila ohjelmistokehityksessä. Löyhä sidoksellisuus saavutetaan asettamalla toisesta luokasta riippuvainen luokka viittaamaan ainoastaan toisen luokan rajapintaan (engl. *class interface*), jolloin viitatun rajapintaluokan pohjalta voidaan toteuttaa erilaisia tilanteen vaatimia sovellutuksia. Luokka sisältää vain sopimuksen siitä, millaisia luokkamuuttujia ja -metodeja viitattu luokka voi sisältää. Tällöin mikä tahansa rajapinnan toteuttava luokka voi esiintyä toisen luokan sisällä. Sidoksellisuuden vähentymisen lisäksi löyhän sidoksellisuuden etuna on parempi sovelluksen laajennettavuus ja päivitettävyys, mitä tiukan sidoksellisuuden toteutuksessa ei voitaisi saavuttaa. [Kaye, 2003]

MVC-pohjaisessa sovelluksessa sidoksellisuuden esiintyvyyteen vaikuttaa keskeisesti mallin, ohjaimen ja näkymän erottelu ja riippumattomuus toisistaan.

Perinteisesti näiden osien riippumattomuuteen käytetään tarkkailijamallia [Purdy & Richter, 2002]. Lisäksi sidoksellisuuden voidaan katsoa vähenevän sellaisissa MVC-ratkaisuissa, joissa näkymän voi vaihtaa ja joissa useiden eri näkymien liittäminen yhteen ohjaimeen tai malliin on mahdollista.

3.5.2. Luettavuus

Luettavuus (engl. *readability*) viittaa sovelluksen koodin ymmärrettävyyden helppouteen sekä sovelluksen dokumenaatioon että ohjelmakoodiin nojautuen. Luettavuuteen voidaan katsoa vaikuttavan usea erillinen sovelluskoodin rakenteelliseen suunnitteluun ja toteuttamiseen liittyvä seikka, kuten ohjelman rakenne, nimeämiskäytännöt, funktiokuvaukset, koodikommentit sekä sovelluksen rakennetta kuvaava dokumentaatio. [Haneef, 1998]

Luettavuutta voi vaikeuttaa epälooginen toimintakokonaisuuksien sijoittelu, saman koodin toisto useassa paikassa, sovelluskehyksen tai sovelluksen sisäisen ohjelmalogiikan säilyttäminen piilossa sovelluskehittäjältä tai liian mutkikkaat tai monitulkintaiset metodien nimeämiskäytännöt. Lisäksi luettavuuteen vaikuttaa luokkien ja metodien pituus ja käytettyjen ohjelmakäskyjen monimutkaisuus [Buse, 2008].

Luettavuuteen vaikuttaa myös metodeissa esiintyvän koodin toteutus. Liian pitkien metodien toiminnallisuus tulisi mahdollisuuksien mukaan jakaa useaan osaan, jolloin metodin toiminnallisuuden hahmottaa helpommin. Luettavuutta voi lisätä myös kommentoimalla koodia [Steidl et al., 2013], mutta liiallisen kommentoinnin tapauksessa voi olla kyse tarpeesta kommentoida liian monimutkaisesti toteutetun koodin rakennetta [Tan, 2012]. Koodi on luettavaa, jos siitä voi saada selvää toinen kehittäjä jo ilman kommentteja.

Lisäksi luettavuuteen vaikuttaa metodien toiminnallisuuksien oikea jakaminen: peruslähtökohtana tulisi olla, että metodi tekee mentaalitasolla vain yhden toimintakokonaisuuden. Mikäli toimintakokonaisuus koostuu useista alatoiminnallisuuksista, tulisi nämä erotella erillisissä sisäisissä metodeissa suoritettaviksi. Lisäksi metodilla ei saisi olla sellaisia sivuvaikutuksia, joita metodin nimeämiskäytännön perusteella ei voida yleisesti ottaen päätellä. Näiden havaintojen pohjalta tarkasteltuna luettavuus muistuttaa käsitteenä lähinnä koodin *ymmärrettävyyttä*. Lisäksi koodin luettavuudella on merkitystä sovelluksen ylläpidettävyyden helppouden kannalta [Buse, 2008].

3.5.3. Tehokkuus

Tehokkuudella (engl. *performance*) mitataan, kuinka nopeasti sovellus suoriutuu sille asetetuista laskutehtävistä käytettyjen resurssien määrään suhteutettuna [ISO-9126, 2000]. Laajemmasta näkökulmasta tarkasteltuna tehokkuus voi

viitata myös sovelluksen käyttöön liittyvään tehokkuuteen, mikä toisaalta on sidoksissa sovelluksen käytettävyyteen, käytön helppouteen ja sovelluksella tehtävän työtehtävän tuotettavuuteen. Kun tarkastellaan sovelluksen tai sovelluskehyksen arkkitehtuuria, tulee kuitenkin erityisesti painottaa sovelluksen rakenteellisesta toteutustavasta koituvaa rasitetta sovelluksen tehokkuudelle. Tällöin on otettava huomioon nimenomaan sovelluksen arkkitehtuurin ja teknisen toteutustavan vaikutukset suorituskykyyn.

Javascript-pohjaisissa MVC-sovellusratkaisuissa tehokkuutta voi heikentää esimerkiksi luokkapohjaisen ohjelmointiparadigman liian tarkka noudattaminen. Lisäksi liiallinen viestittely sovelluksen MVC-osien välillä saattaa hidastaa sovelluksen suorituskykyä. Tässä tapauksessa on harkittava ratkaisua suorituskyvyn ja ymmärrettävän arkkitehtuurin väliltä: onko lyhyestä viipeestä haittaa sovelluksen suorituskyvylle ja käytettävyydelle kyseenomaisen toteutuksen tapauksessa, kun sovelluksen sisäisten viestintäratkaisujen ja ohjelmointiparadigmakohtaisen arkkitehtuurin avulla voidaan sovelluskoodista saada luettavampaa ja ylläpidettävämpää.

Lisäksi erityisesti verkkosovelluksissa tehokkuuteen vaikuttaa oleellisesti käyttöliittymän sisäinen toteutus: on harkittava, hidastaako sivustoa näkymäpohjien erillinen lataaminen palvelimelta oleellisesti sovelluksen käytettävyyttä vai onko suorituskyvyn kannalta edullisempaa päätyä ratkaisuun, jossa käyttäjälle tarjotaan heti ensimmäisen sivulatauksen yhteydessä valmis, palvelimella koostettu lähtökohtainen näkymän rakenne. Tällöin tehokkuuteen vaikuttavissa tekijöissä on otettava huomioon erityisesti palvelimen ja asiakkaan väliseen viestittelyyn liittyvät arkkitehtuurilliset ratkaisut.

3.5.4. Siirrettävyys

Siirrettävyydellä (engl. *portability*) kuvataan mahdollisuutta siirtää sovellus ympäristöstä toiseen. Ympäristö voi liittyä organisaatioon, laitteistoon tai ohjelmistoon. [ISO-9126, 2000].

Oliopohjaisissa ratkaisuissa esiin nousee erityisesti luokkien uudelleenkäytettävyys. Tässä mielessä siirrettävyyteen vaikuttaa oleellisesti ohjelmakoodin sidoksellisuus. Löyhän sidoksellisuuden ansiosta myös siirrettävyys paranee: ohjelman eri osia voi todennäköisemmin käyttää muiden sovellusten toiminnallisuusvaatimusten täyttämiseen.

Siirrettävyydessä on kyse myös sovelluksen mukautettavuudesta toisessa alustassa käytettäväksi. Verkkopohjaisissa sovelluksissa alustana on verkkoselain, joten sovelluksen siirrettävyys ilmenee etenkin selaintuen kattavuuden laajuudessa.

3.5.5. Ylläpidettävyys

Ylläpidettävyydellä (engl. *maintainability*) kuvataan mahdollisuutta muuttaa sovelluksen toiminnallisuutta. Ylläpidettävyys kertoo sovelluksen kyvystä säilyttää toimintakyky muuttuneiden olosuhteiden aikaansaamien muutos- ja lisäystarpeiden myötä [ISO-9126, 2000].

Verkkosovelluksen ylläpidettävyyteen vaikuttavat useat eri tekijät. Usein ohjelmakoodin kanssa on tekemisissä useampi ohjelmistokehittäjä, joten ylläpidettävyyden kannalta on otettava huomioon erityisesti se, että mahdolliset asiakkaiden tai muiden sidosryhmien asettamat uudet ominaisuus- ja muutosvaatimukset voidaan täyttää sovelluksen toimivuuden tästä kärsimättä. Tässä ohjelmistokoodin laadulla on merkittävä vaikutus. On myös harkittava, onko yksittäiseen sovelluskoodin tiedostoon tai luokkaan sisällytetty liikaa koodia, ja onko muiden ohjelmoijien mahdollista ymmärtää ja muokata ohjelmakoodia vaivattomasti. Tämän takia erityisesti koodin luettavuudella ja laajennettavuudella on ylläpidettävyyden kannalta suuri merkitys.

3.5.6. Laajennettavuus

Laajennettavuus (engl. *extendability*) ilmenee sovelluksen muokattavuutena sekä kykynä lisätä toiminnallisuutta sovellukseen.

Jotkin verkkosovelluskehykset mahdollistavat itse kehyksen toiminnan laajentamisen erinäisin lisäosin. Toisaalta sovelluskehys saattaa olla rakennettu alusta asti modulaarisesti, jolloin laajennettavuus ilmenee mahdollisuutena lisätä ja poistaa sovelluskehyksen osia sovelluskohtaisten tarpeiden mukaan.

Verkkosovelluksissa laajennettavuus ilmenee niin ikään mahdollisuutena mukauttaa paitsi sovelluskohtaisia myös sovelluskehyksen sisäisiä luokkia, käyttämällä luokkaperintää tai muuta laajennettavuutta tukevaa ratkaisua hyödyksi.

3.5.7. Koko

Koko (engl. *size*) kertoo sovelluksen tai sovelluskehyksen ohjelmakoodin sekä oheiskirjastojen viemän tilan tavuina. Sovelluksen ohjelmakoodin koko ei sinällään kerro sovelluksen tehokkuudesta mitään, mutta se voi tietyissä tilanteissa vaikuttaa sovelluksen suorituskykyyn. Verkkosovellusten tapauksessa sovelluskehyksen ja sovelluskehyksellä tehdyn sovelluksen koko vaikuttaa verkkosovelluksen sivulatauksen nopeuteen. Tällä voi olla merkitystä etenkin mobiilisovelluksien kehittämisen kannalta, mobiililaitteiden mahdollisesti hitaamman tiedonsiirtonopeuden vuoksi.

Sovellusohjelman kokoa voidaan mitata myös koodirivimäärän, ohjelmakoodin kompleksisuuden, luokkafunktiomäärän ja luokkien kokonaismäärän perusteella [Tan, 2012]. Sovelluskehyksellä toteutettujen sovellusten tapauksessa näillä voi olla kuitenkin myönteinen vaikutus itse sovelluksen ohjelmakoodin kokoon: kun sovelluksen vaatimia toiminnallisuuksia voidaan suorittaa sovelluskehyksen tarjoamien rajapintojen avulla, sovelluksen ohjelmakoodin koko laskee.

Sovelluskehyksien kokoa voi kasvattaa myös niin kutsutun *paisuneen koodin* (engl. *bloat code*) ilmeneminen [Uhlig, 1995]. Laajasti käytettyjen verkkosovelluskehysten on usein tuettava niin ikään kehyksen aikaisimmilla versioilla tehtyjä sovelluksia, jolloin vanhaa koodia on säilytettävä taaksepäin yhteensopivuuden säilyttämiseksi.

Toisaalta muutamat verkkosovellusten ohjelmointiin käytettävistä sovelluskirjastoista ovat karsineet vähitellen pois vanhempien selainten ja standardien edellyttämiä ohjelmointirajapintoja tiedostokoon optimoinnin nimissä. Tällöin tuki vanhemmille rajapinnoille saatetaan tarjota erikseen ladattavan, vanhaa sovelluskoodia tukevan komponentin avulla, kuten jQuery-kirjasto on vastikään tehnyt [jQuery, 2014]. Tällöin uusien sovellusten suorituskykyä ei jouduta rasittamaan turhaan vanhan koodikannan tukemisen takia.

4. Sovelluskehysten esittely

Tässä luvussa esittelen tutkielmaani valitsemani sovelluskehykset. Esittelen kehysten peruspiirteet sekä tavallisimmat tavat luoda yksinkertaisia MVC-pohjaisia sovelluksia. Myöhemmin luvun 5 arvioinnissa ja vertailussa tuon esiin näiden kehysten avulla luotavien sovellusten laatuominaisuusvahvuuksia sekä yleisellä tasolla tuotettavan koodin keskeisimmät luonteenpiirteet.

Sovelluskehysten valintakriteerinä on ollut kehyksen tunnettuus sekä arkkitehtuurisesti kiinnostavat piirteet kehyksessä. Esiteltävinä ovat Maria Framework, jQueryMX ja KnockoutJS. Maria Framework on mukana tarkastelussa siksi, koska sen arkkitehtuurilliset tavoitteet näyttävät poikkeavan filosofialtaan oleellisesti muista kehyksistä. Toisaalta jQueryMX on mukana siksi, koska sen arkkitehtuuri pohjautuu laajimmin tunnettuun Javascript-pohjaiseen DOM-rakenteenmuokkaustyökaluun, jQuery-kirjastoon. Lisäksi KnockoutJS on muista kehyksistä arkkitehtuuriltaan poikkeavana mielenkiintoinen, sillä siinä kommunikointi MVC-mallin osien välillä tapahtuu tietoliitoksia käyttäen, eikä perinteistä imperatiivista ohjelmointiparadigmaa hyödyntäen.

4.1. Maria Framework

Maria on Peter Michaux'n vuonna 2012 Javascriptille kehittämä MVC-pohjainen sovelluskehys, joka pyrkii jäljittelemään aitoa alkuperäistä Smalltalk-kielelle 1970-luvulla esiteltyä MVC-mallia. Tekijä kutsuu kehystä puhtaaksi MVC-ratkaisuksi, sillä siinä, toisin kuin valtaosassa muita Javascript-sovelluskehyksiä, annetaan ohjaimelle oma tarkoin määritelty käyttötarkoituksensa. [Michaux, 2013]

Ohjain ei siis sisällä malliin liittyvää koodia eikä malli sisällä ohjaimeen liittyvää koodia. Niin ikään ohjaimessa ei käsitellä näkymään liittyviä toiminnallisuuksia, vaan näkymäluokka on tarkoitettu ainoastaan tiedon esittämiseen liittyvään toiminnallisuuteen. Kehyksessä MVC-arkkitehtuuri onkin jaettu seuraaviin tehtäviin [Michaux, 2013]:

1. Malli sisältää datan. Malli tiedottaa tilansa muuttumisesta sitä tarkkaileville olioille.
2. Näkymä tarkkailee mallia ja esittää graafisesti mallinsa tämän hetkisen tilan. Näkymällä on yksi ohjain. Näkymällä voi olla useita alinäkymiä.
3. Ohjain päättää mitä tehdään, kun käyttäjä on vuorovaikutuksessa ohjaimen näkymän kanssa.

Kehyksen kehittäjä mainitsee malli-, näkymä- ja ohjainolioiden käyttävän avukseen myös kolmea muuta keskeistä MVC-malliin liittyvää suunnittelumallia: *tarkkailija-tarkkailtava-suunnittelumallia, komposiittimallia* ja *strategiamallia*.

Näiden kolmen suunnittelumallin käytön lisäksi Maria-kehyksessä käytetään perinteisiä MVC-toteutuksen malleja: tehdasmetodimallia (engl. *factory method pattern*) ja työpohjia.

Kaikki kehyksen omat oliot ovat *maria*-nimisen nimiavaruuden alla. Olioita ovat muun muassa *Model, Controller* ja *View*. Lisäksi luotavalle sovellukselle voidaan luoda oma nimiavaruus. Tähän sovelluksen nimiavaruuteen liitetään oliot, jotka luodaan Maria-kehyksen nimiavaruuden alla olevien olioiden metodien avulla. Luokka luodaan Maria-kehyksessä laajennettavan luokan *subclass*-metodilla (ks. listaus 7).

```
maria.Model.subclass(ohjelmanNimiavaruus, 'OmaMalli', {
    properties: {
      nimi: 'oletusarvo',
      setNimi: function(nimi) {
        this.nimi = nimi;
      },
      getNimi: function() {
        return this.nimi;
      }
    }
});
```

Listaus 7. Malli luodaan ohjelman nimiavaruuteen kehyksen *Model*-olion *subclass*-metodilla.

Maria-kehyksessä jokaiselle käyttöliittymän elementille voi määritellä oman näkymäluokkansa. Luokasta *ElementView* tehdään ilmentymä *subclass*-metodilla (ks. listaus 8). Käyttöliittymän elementteihin kohdistuvat käyttöliittymätapahtumat liitetään ohjaimen funktioihin näkymän *uiActions*-taulukolla. Näkymää vastaava HTML-dokumentissa esitettävä DOM-puu rakennetaan *properties*-taulukon *buildData*-funktiossa. Malli kutsuu näkymän *update*-funktiota, kun mallia muutetaan. Tällöin *update*-funktio päivittää näkymän sisällön kutsuen näkymän *buildData*-funktiota.

```
maria.ElementView.subclass(ohjelmanNimiavaruus, 'OmaView', {
    uiActions: {
        'click .painike': 'onClickButton'
    },
    properties: {
        buildData: function() {
            var model = this.getModel();
            this.find('.nimiDiv').innerHTML = model.getNimi();
        },
        update: function() {
            this.buildData();
        }
    }
});
```

Listaus 8. Käyttöliittymäkomponentin näkymää ohjataan *ElementView*-oliolla.

Näkymän ja mallin lisäksi tarvitaan ohjain, joka näkymän kutsumana päivittää mallin tilaa (ks. listaus 9).

```
maria.Controller.subclass(ohjelmanNimiavaruus, 'OmaController', {
    properties: {
        onClickButton: function(nimi) {
            this.getModel().setNimi(nimi);
        }
    }
});
```

Listaus 9. Ohjain sisältää luettelon mahdollisista tapahtumakäsittelijöistä.

Ohjaimessa määritellään mahdolliset käyttöliittymätapahtumat funktioina *properties*-taulukon sisällä. Jos funktiolle annetaan parametreja, kuten listauksen esimerkissä on tehty, tulee käyttöliittymätapahtuman funktiota vastaava funktio määritellä myös näkymässä. Tällöin näkymä voi hakea tarvittavan parametrin (tässä *nimi*-muuttujan arvon) esimerkiksi käyttöliittymän HTML-lomakkeen tekstikentästä.

Maria-kehyksen luokkien nimeämiskonvention ansiosta Maria-kehys osaa automaattisesti, vaikka kehittäjä ei näin eksplisiittisesti määrittelisi, liittää ohjaimen nimen perusteella ohjaimeen vastaavan mallin ja sovelluksen näkymään näkymää vastaavan ohjaimen.

Maria-kehyksessä ohjain vaikuttaa olevan tietyssä määrin vain näennäisesti erotettu näkymästä. Näkymä liittää HTML-elementtiin kohdistuvat käyttöliittymätapahtumat niitä vastaaviin ohjaimen funktioihin. Tällöin näkymä on

vastuussa käyttöliittymätapahtumien välittämisestä ohjaimelle, jolloin näkymä muuttuu aktiiviseksi toimijaksi ja ohjain toimintojen passiiviseksi suorittajaksi.

Toisaalta, koska Maria-kehyksessä ohjain ei suoraan päivitä näkymää, vaan näkymä päivittyy mallin tilan muuttamisen kautta, toteutuu MVC-mallin alkuperäinen "sirkulaarinen" semantiikka: Käyttöliittymätapahtuman myötävaikuttamana näkymä kutsuu tilanteeseen soveltuvaa ohjaimen funktiota. Ohjaimen funktio muuttaa vastaavasti mallin tilaa. Mallin tilan muuttumisen myötä malli ilmoittaa tarkkailijoilleen, kuten näkymälle, muutoksesta, jolloin näkymä päivittää HTML-rakennetta mallin nykyisen tilan mukaisesti.

Maria-kehys vaikuttaa toteutukseltaan hyvin supistetulta. Sen mukana ei tule mitään ylimääräisiä käyttöliittymäkomponenttikirjastoja, vaan kehyksen keskeisin tavoite näyttää olevan tarjota sovelluskehittäjille puitteet sovelluksen yleisten toiminnallisuuksien, kuten tapahtumienkäsittelyn, rakentamiseen "oikeaa" MVC-mallia käyttäen. [Osmani, 2012a]

Toisaalta Maria-kehyksen toimintaa voi laajentaa erillisin kehyksen kanssa yhteensopiviksi suunnitelluin lisäosin (engl. *plugin*). Näin kehyksellä kehitettävän sovelluksen toiminta ei rajoitu kehyksen mukana valmiina tarjottuihin toiminnallisuuksiin.

Lisäksi Maria-kehys näyttää käyttävän melko paljon konfiguroinnin sijasta konventiota (engl. *convention over configuration*). Tällöin osa sovelluksen toiminnallisuudesta tapahtuu kehittäjältä "piilossa"; kehittäjän on vain tiedettävä, milloin jokin toiminnallisuus tapahtuu, vaikka kyseistä sovelluskehyksen ominaisuutta tai toiminnallisuutta ei ole eksplisiittisesti kutsuttu sovelluskehyksellä tehdystä sovelluskoodista käsin. Toisaalta tästä on etua, sillä näin voidaan muun muassa sovelluskehityksen alkuvaiheessa jättää vähemmän tärkeitä vaatimusmäärittelyn osa-alueita toteuttamatta, kun sovelluskehys tekee nämä toiminnallisuudet automaattisesti.

Tällä hetkellä Maria-kehykselle ei ole olemassa kunnon dokumentaatiota, joten kehyksen käyttö on opeteltava ohjelmakoodissa olevia kommentteja, API-dokumentaatiota ja Marian sivustolla olevia ohjelmaesimerkkejä seuraten. Tätä voitaneen kuitenkin pitää vain väliaikaisena ongelmana, sillä näyttää, että sovelluskehystä ja sen dokumentaatiota päivitetään ja kehitetään edelleen.

4.2. jQueryMX

JQueryMX on osa *JavascriptMVC*-nimistä sovelluskehystä, jonka väitetään käyttävän MVC-arkkitehtuuria. JQueryMX koostuu muutamasta jQuery-kirjastosta, joilla laajojen jQuery-pohjaisten sovellusten toteuttaminen ja järjestäminen onnistuu hallitusti.

JQueryMX jaetaan neljään päalueeseen: *DOM-avustimet, kieliavustimet ja erikoistapahtumat* sekä *malli-, näkymä-, ohjain- ja luokkarakenteet.* Kehyksen MVC-mallin toteuttavat luokat *$.Model, $.View* ja *$.Controller* vievät yhteensä tilaa vain 7 kilotavua. [JavaScriptMVC, 2014]

DOM-avustimet tuovat mukanaan sellaisia usein tarvittavia toimintoja ja DOM-muokkausfunktioita, jotka puuttuvat jQueryn perusasennuksesta. Näitä ovat muun muassa selaimen evästeiden käsittely, kehittyneempi Ajax-hallinta sekä laajennetut DOM-rakenteen hakufunktiot.

Erikoistapahtumat lisäävät sovelluksen vuorovaikutteisuutta. Laajennuksen mukana tulee muun muassa yksinkertaistettu tuki raahauksen, elementtien kohdistuksen, koon muuttamisen sekä käyttöliittymän tapahtumien tilanne-kohtaisen keskeyttämisen ja jatkamisen toteuttamiseen.

Kieliavustinten avulla laajennus helpottaa erinäisin funktioin Javascript-datan käsittelyä. Avustin laajentaa sellaisia suunnittelumallien toteutuksia, jotka joissain kielissä tulee automaattisesti kielen oletuskirjastojen mukana. Avustimiin kuuluvat *Object-, Observer-, String-, toJSON-* sekä *Vector*-luokat. Näistä *Observer*-luokka toteuttaa tarkkailijamallin laajennusta käyttäen.

MVC-toteutuksen ohjain ja malli perivät *$.Class*-luokan ominaisuudet. Luokka tarjoaa sovellusten käyttöön yksinkertaisen prototyyppisen kielen luokkaperinnän. [JavascriptMVC, 2014]

Luokkaperinnässä staattiset metodit ja prototyyppimetodit määritellään omana taulukkolohkonaan (ks. listaus 10) ja olion ilmennys toteutetaan prototyyppimetodina *init()*.

```
$.Class("Opiskelija",
  { //Staattisia muuttujia käytetään luokan kaikista ilmentymistä.
    ammatti: "Opiskelija"
  },
  { //Prototyypin metodit:
    init: function(etunimi, sukunimi) {
      this.etunimi = etunimi;
      this.sukunimi = sukunimi;
    },
    kokoNimi: function() {
      return this.etunimi + " " + this.sukunimi;
    }
  }
);

var opiskelija = new Opiskelija("Jouni", "Kähkönen");
console.log( opiskelija.kokoNimi() );
```

Listaus 10. Luokka määritellään jQueryMX-kehyksen luokkamallin mukaisesti, mutta ilmentymän voi luoda tavallisesti Javascriptin omalla *new*-operaattorilla.

JQueryMX-kehyksen malli on toteutettu siten, että se voi hakea tiedon vaihtoehtoisesti suoraan palvelimelta. Mallissa on toteutettuna metodit *findAll, findOne, create, update* ja *destroy* tiedon hakua ja muokkausta varten. Metodi *findAll* hakee kaikki rivit palvelimelta, metodi *findOne* hakee yhden id-tunnuksen mukaisen rivin palvelimelta. Metodi *create* luo uuden rivin, *update* muuttaa rivin sisältöä ja *destroy* poistaa rivin palvelimelta ja mallista. [JavaScriptMVC, 2014]

Kommunikointi käyttöliittymän ja mallin välillä toteutetaan malliolion *bind()*-metodia käyttäen. Metodille annetaan parametrina tarkkailtavan attribuutin nimi sekä toisena parametrina funktio, joka suoritetaan, kun attribuutin arvo muuttuu (ks. listaus 11).

```
opiskelija.bind("etunimi", function(event, uusiNimi) {
  $(body).html('Nimi: ' + uusiNimi);
});
```

Listaus 11. Tapahtumia valvotaan JQueryMX-kehyksessä mallin *bind()*-metodin avulla.

Uusien malliolioiden luontia tarkkaillaan vastavuoroisesti malliluokan *bind()*-metodilla (ks. listaus 12): metodille annetaan parametriksi merkkijono "*created*" ja toisessa parametrissa annettavan funktion ensimmäinen parametri sisältää viittauksen tapahtumaolioon ja toinen parametri viittauksen vasta luotuun olioon.

```
opiskelija.bind("created", function(event, uusiOpiskelija) {
  $(body).append('Uusi opiskelija: ' + uusiOpiskelija.kokoNimi());
});
```

Listaus 12. Uuden olion luontia valvova tapahtumafunktio.

Varsinaista näkymäoliota jQueryMX-kehyksellä tehdyssä sovelluksessa ei ole; olio *$.View* on vain työpohjakehys. Työpohjat ovat yksinkertaisia lyhyitä HTML-muotoista merkkijonodataa sisältäviä tiedostoja, joissa on paikanvaraajat kulloisenkin mallin datalle. Näkymän käytön sijaan jQueryMX-kehyksessä on menettelytapa työpohjan hakemiseksi palvelimelta tai suoraan erillisestä Javascript-muuttujasta. Varsinaisena näkymänä jQueryMX-kehyksellä tehdyssä sovelluksessa pidetään työpohjaa tai laajemmassa mittakaavassa koko sovelluksen HTML-rakennetta.

Lista opiskelijoista voitaisiin esimerkiksi hakea mallin *findAll*-metodilla palvelimelta ja sitten päivittää näkymään työpohjaa hyödyksi käyttäen (ks. listaus 13).

```
Opiskelija.findAll({}, function(opiskelijat) {
  $('#opiskelijat').html( 'tyopohjat/opiskelijat.ejs' );
```

Listaus 13. Näkymä päivitetään *.ejs*-muotoisen työpohjan avulla.

JQueryMX-kehyksen ohjain *$.Controller* on lisäosien luontiin käytettävä tehdasmetodi. Perinteisen MVC-mallin mukaan näkymä on ohjaimen jäsen, eli näkymällä on aina jokin ohjain, jossa näkymään liittyvä tapahtumankäsittely tapahtuu. Näin myös jQueryMX-kehyksen ohjain liitetään haluttuun näkymään. Koska jQueryMX-kehyksessä näkymä on sivuston HTML-rakenteen osa, liitetään ohjain haluttuun DOM-puun osaan jQuery-laajennuksesta tutun menettelytavan mukaan. Näin voidaan määrittää yleispäteviä tapahtuman-käsittelyliitoksia HTML-elementteihin. Esimerkiksi liitoksella *"LI click"* liitetään napsautustapahtuma ohjaimen näkymän LI-listaelementtiin tai -elementteihin. Tietyn elementin sijasta voidaan tarkkailla myös mitä tahansa ohjainluokan määrettä asettamalla muuttujan nimi aaltosulkeisiin: esimerkiksi liitoksen *"{opiskelija} created"* avulla tapahtumafunktio liitetään luokan *opiskelija*-nimisen mallin lähettämään *created*-tapahtumaan. [JavaScriptMVC, 2014b]

Kehyksen määrittelemän tapahtumatuen avulla on mahdollista tehdä käyttöliittymäkeskeisiä komponentteja. Toiminnallisuuden joustavuutta korostavan toteutustapansa ansiosta ohjain voi toimia joko sovelluksen näkymänä tai perinteisenä ohjaimena. Näkymäkeskeinen ohjain voi toimia käyttöliittymän yksittäisen osan luontikomponenttina rakentaen näin esimerkiksi sivuston valikkorakenteen tai monisivuisen taulukon näkymän. Toisaalta ohjain voi toimia perinteisen ohjaimen tapaan käyttöliittymän ja tietomallien välikappaleena. [JavaScriptMVC, 2014b]

JQueryMX-kehyksen ohjaimessa esitettävät tapahtumafunktiot liitetään käyttöliittymän elementteihin käyttäen yleispäteviä luokkamääreitä. Ainoastaan itse ohjain liitetään suoraan johonkin tiettyyn käyttöliittymän elementtiin. Tämäkin liitos tehdään kuitenkin ohjaimen ulkopuolella, joten ohjaimen uudelleenkäyttö muissa käyttöliittymän elementeissä on mahdollista. Tällöin ohjain ei ole enää sidoksissa yksittäiseen käyttöliittymän elementtiin.

Ohjainluokka perii kehyksen yleisen luokkaolion ominaisuudet, joten käytettävissä ovat staattiset muuttujat, prototyypin metodit sekä *init*-rakenninmetodi, aivan kuten $.Class-luokassakin. Ohjaimen ilmennyksen aikana luodaan lisäksi ohjaimeen perustuva jQuery-avustinfunktio, joka sidotaan olion luonnin aikana haluttuun DOM-puun elementtiin. [JavaScriptMVC, 2014b]

Ohjaimen avulla mallin muutosten tarkkailu helpottuu, kun tapahtuma-funktiota ei tarvitse eksplisiittisesti liittää malliin sen *bind*-metodia käyttäen. Sen sijaan ohjain asetetaan tarkkailemaan mallin muutosta määrittelemällä määreluettelon avaimessa mallin nimi sekä tarkkailtava tapahtuma. Listauksessa 14 kuvataan, kuinka ohjain tarkkailee mallin luontia ja miten ohjain saa päivitettyä luodun rivin työpohjaa käyttävään näkymään. [JavaScriptMVC, 2014c]

```
// Luodaan ohjain.
$.Controller("Opiskelijalista", {
  defaults: {
    template: null,
    model: null
  }
}, {
  init: function() {
    var opiskelijat = this.options.model.findAll();
    this.element.html(this.options.template, opiskelijat);
  },
  "{model} created": function(Model, event, uusiKohde) {
    this.element.append(this.options.template, [uusiKohde]);
  }
});

// Luodaan ohjaimen ilmentymä.
$('#opiskelijat').opiskelijalista({
     model: Opiskelija,
     template: "opiskelijat.ejs"
});
```

Listaus 14. Ohjaimen, näkymän ja mallin välinen viestintä jQueryMX-kehyksen mukaisessa verkkosovelluksessa. [JavaScriptMVC, 2014c]

Listauksen 14 *init*-rakentimessa haetaan mallilta alkiolista käyttämällä mallin *findAll*-metodia hyödyksi. Tämän jälkeen ohjainta vastaava näkymä sovelluksen HTML-rakenteessa päivitetään käyttämällä pohjana ohjaimen *template*-muuttujan mukaista työpohjaa (*this.options.template*) sekä vasta haettua opiskelija-alkioiden luetteloa. Tämän jälkeen ohjaimessa määritellään mallin rivien lisäämistä tarkkaileva tapahtumafunktio. Tapahtumafunktiolle annetaan parametrina itse malli, käyttöliittymätapahtuman tilannekohtaisia tietoja sisältävä *event*-olio sekä malliin lisätty uusi kohde. Esimerkissä päivitetään näkymää siten, että vasta luotua oliota vastaava näkymä on nähtävissä käyttöliittymässä. Tässä on tärkeää huomioida, että ohjainta vastaavaa näkymää HTML-rakenteessa ei päivitetä kokonaisuudessaan, vaan pelkästään uutta kohdetta vastaava näkymän osa päivitetään näkymään. Tähän käytetään

jQuery-kirjaston *append*-metodia, joka lisää annettua taulukkoa vastaavan työpohjan sisällön suoraan ohjainta vastaavan käyttöliittymäelementin loppupäähän.

Vastaavasti mallin kohteiden muuttumisen ja poistamisen tarkkailuun käytetään *updated*- ja *destroyed*-tapahtumia [JavaScriptMVC, 2014b]. Tällöin tapahtumafunktiolle vain annetaan päivitetty tai poistettu kohde parametrina. Ohjaimessa esitettävän tapahtumafunktion tehtävänä on suorittaa tapahtumaa vastaava toimenpide siten, että näkymä pysyy mallin uutta tilaa vastaavana. Tällöin esimerkiksi mallin rivin poiston yhteydessä jää ohjaimen tehtäväksi poistaa vastaava elementti HTML-rakenteesta.

Kun ohjaimen pohjalta luodaan ilmentymä, ohjaimen tehdasmetodille annettavan taulukon määreiden arvot päivitetään ohjaimen *defaults*-taulukossa määritettyjen määreiden arvojen tilalle. Muuttujiin, joita ei ole määritelty tehdasmetodikutsussa, käytetään ohjaimen *defaults*-taulukon arvoja. JQuery-kirjaston tehdasmetodityyppisen olioilmennyksen sijasta olio voidaan vaihtoehtoisesti ilmentää perinteisesti *new*-operaattorin avulla. Ohjaimen yläluokan rakentimesta käsin alustetaan jo kaikki ohjaimen tapahtumakäsittelyyn liittyvä toiminnallisuus, joten sovelluskehittäjä voi itse valita, alustaako olion *new*-operaattorilla vai tehdasmetodia hyödyntäen.

Esimerkeistä nähdään, että jQueryMX-kehyksessä ohjaimen tehtävänä on käyttöliittymätapahtumien käsittelyn lisäksi mallin nykyisen tilan päivittäminen suoraan sovelluksen käyttöliittymään. Vaikka ohjaimessa esiintyvää käyttöliittymäkohtaista koodia saadaan vähennettyä huomattavasti käyttämällä työpohjia sovelluksen käyttöliittymän rakentamisessa, ilmestyy ohjaimeen helposti näkymäriippuvaista koodia muun muassa tiedon päivittämisen, poiston ja lisäämisen yhteydessä.

4.3. KnockoutJS

KnockoutJS on Microsoftin työntekijän Steve Sanderssonin kehittämä vapaan lähdekoodin Javascript-kehys, joka käyttää *Model-View-ViewModel*-suunnittelumallia. KnockoutJS-kehyksen kantavana voimana on kaksi pääperiaatetta: a) datan, näkymäelementtien sekä datan esityksen pitää olla selkeästi erotettu toisistaan, ja b) näkymäelementtien väliset suhteet on eriytettynä omaan luokkaansa. [KnockoutJS, 2013]

KnockoutJS-kehyksessä tulee mukana myös natiivi tuki työ- eli sivupohjien käytölle. Kehys ei kuitenkaan pakota käyttämään natiivia sivupohjatukea, vaan kehittäjä voi vapaasti käyttää näkymän rakenteen esittämiseen kolmannen osapuolen tarjoamaa sivupohjakirjastoa. [KnockoutJS, 2013]

Kehyksessä elementteihin voidaan liittää tiettyihin käyttöliittymätapahtumiin liittyviä tietoliitoksia. Tällöin käyttöliittymästä saadaan helposti itsestään päivittyvä. Kun tietomallin tilaa eli mallin jonkin muuttujan arvoa muutetaan, käyttöliittymä päivittää nämä muutokset automaattisesti näkymään.

Automaattisesta päivityksestä on hyötyä, kun tehdään laajempia selainpohjaisia sovelluksia. Tällöin kehittäjän ei tarvitse luoda koodia, jossa muuttujiin tai taulukoihin tehdyt muutokset päivitettäisiin erikseen näkymään. Riittää, kun esimerkiksi lisää uuden elementin taulukkoon, jolloin muutokset ilmestyvät automaattisesti niihin kohtiin HTML-näkymää, joista taulukko on merkitty riippuvaiseksi liitosten avulla.

Näkymänä toimii KnockoutJS-kehyksellä tehdyssä sovelluksessa käyttöliittymän HTML-rakenne, kun taas näkymämallina toimii tavallinen Javascript-olio, johon on liitetty tarkkailtavia attribuutteja. Malli ei ole erillinen olio, vaan se koostuu näkymämallin valvomista attribuuteista, joiden tarkkailu käynnistetään erillistä kehyksen metodia kutsumalla (ks. listaus 15).

```
<!-- Näkymä eli HTML-rakenne: -->
<p>Etunimi: <input data-bind="value: firstName" /></p>
<p>Sukunimi: <input data-bind="value: lastName" /></p>
<h2>Hei, <span data-bind="text: fullName"></span>!</h2>

<!-- Näkymämalli: -->
<script>
  function MyViewModel() {
      this.firstName = ko.observable("Jouni");
      this.lastName = ko.observable("Kähkönen");
      this.fullName = ko.computed(function() {
          return this.firstName() + " " + this.lastName();
      }, this);
  }
  ko.applyBindings(new MyViewModel());
</script>
```

Listaus 15. KnockoutJS-kehyksen mukainen HTML-näkymä sekä tietoliitoksista koostuva näkymämalli. [KnockoutJS, 2013]

Listauksen 15 näkymässä määritetään kaksi tekstikenttää ja tekstikappale. Ensimmäisen tekstikentän arvo on liitetty *"value:"*-määritelmällä mallin *firstName*-parametriin ja toinen tekstikenttä *lastName*-parametriin. Näkymämallissa luodaan kaksi tarkkailtavaa muuttujaa (engl. *observable*) sekä yksi laskennallinen funktio *fullName*, jonka arvo määräytyy parametrina annetun funktion paluuarvon perusteella.

Näkymämalli tietoliitoksineen otetaan käyttöön ohjelmassa KnockoutJS-kehyksen mukana tulevalla *ko.applyBindings*-metodilla. Metodille annetaan parametrina ilmentymä MyViewModel-luokasta, jolloin metodi havaitsee mallissa määritellyt tarkkailtavat muuttujat ja funktiot. Tässä yhteydessä myös alkuarvot asetetaan automaattisesti HTML-elementteihin. [KnockoutJS, 2013]

Jos tekstikenttien arvoja muutetaan, muutokset välitetään automaattisesti tietomallin *firstName*- ja *lastName*-muuttujille. Vastaavasti jos muuttujien arvoja muutetaan tietomallista käsin, muutokset päivittyvät automaattisesti tekstikenttiin.

KnockoutJS-kehyksellä tehty sovellus ei sisällä ollenkaan ohjainta. Ohjaimen tilalla on näkymämalli, jonka tehtävänä on sekä määritellä ohjelman data että ohjelman käyttöliittymän käyttäytyminen.

5. Sovelluskehysten arviointi ja vertailu

Tämä luku keskittyy esiteltyjen sovelluskehysten arviointiin ja vertailuun. Aluksi esitellään kehysten arkkitehtuuria yleisellä tasolla. Tämän jälkeen arvioidaan kunkin kehyksen yleistä arkkitehtuuria sekä keskeisiä ohjelmakoodin laatuun vaikuttavia tekijöitä. Arvioinnin jälkeen kehysten arkkitehtuuria vertaillaan eri laatuominaisuuksien mukaan. Lopuksi esitetään vertailun lopputulos ja yhteenveto.

Eri kehysten arvioinnin ja vertailun pohjalta havaitaan kunkin kehyksen hyvät puolet ja toisaalta pyritään löytämään mahdollisia arkkitehtuurillisia ja muunlaisia heikkouksia.

5.1. Yleistä kehysten arkkitehtuurista

Lähes kaikki vertaillut sovelluskehykset toteuttavat oliomaailman luokkien käsittelyn omalla tavallaan. Tästä on paljon haittaa. Sovelluksen koodikannan hyödyntäminen toisessa ympäristössä hankaloituu, koska sovelluksen osien koodi on riippuvainen kehyksen rakenteesta. Toisaalta oliomaailman käsitteiden toteuttaminen sovelluskehyksissä on ymmärrettävää; puuttuuhan Javascriptistä tuki luokkien käytölle.

Tiettyä sovelluskehystä käyttävän koodin sovittaminen toisessa projektissa käytettäväksi vaatii koodissa käytettyjen luokkien ja sovelluskehyskohtaisten rajapintakutsujen muuntamista kohteena olevan projektin sovelluskehyksen mukaisiksi. Tällainen koodin sovittaminen on entistä hankalampaa, jos kohdeprojekti käyttää eri MVC-mallin johdannaista kuin alkuperäinen koodi.

Sovelluskehysriippuvaisen koodin käytöstä voi olla muutakin haittaa. Jos sovelluskehys pakottaa kehittäjän käyttämään esimerkiksi kehitettävän ohjelman MVC-rakenteen malleissa kehyskohtaista luokkarakennetta ja kehysfunktiorajapintakutsuja, on ennestään olemassa olevan koodin hyödyntäminen tällaista kehystä käyttävän sovelluksen kohdalla vaikeaa. Koodi on sovitettava kohteena olevan sovelluksen käyttämän sovelluskehyksen luokkarakenteita vastaavaksi, mikä saattaa aiheuttaa projektista riippuen jopa enemmän työtä kuin vastaavan koodin uudelleenkirjoittaminen kohdesovelluskehyksen rakenteen mukaisena.

MVC-kehyksen käyttö saattaa monimutkaistaa sovelluksen arkkitehtuuria, mutta kehyksen käytöllä voi kuitenkin olla lukuisia etuja verrattuna yksinkertaisten DOM-muokkauskirjastojen kuten jQueryn käyttöön. DOM-muokkauskirjastoillakin voi huomattavasti helpottaa verkkosivujen toteuttamista, mutta niistäkään ei ole välttämättä hyötyä, kun on tarkoitus rakentaa

todellisia, selainpohjaisia ohjelmistoja. Tällaisten ohjelmistojen valmistuksessa sivuston vuorovaikutus käyttäjän kanssa lisääntyy ja sivusto joutuu olemaan yhteydessä palvelimeen reaaliajassa. Ilman kunnollista MVC-pohjaista sovelluskehystä ohjelmakoodista tulee helposti sekavaa ja rakenteeltaan epäjärjestelmällistä. Tällöin koodin ylläpidettävyys ja testattavuus kärsivät. Kun käytetään MVC-pohjaista sovelluskehystä, sovelluksen hallittavuus ja myöhemmät päivitykset sovelluksen toimintaan ovat helpommin toteutettavissa. [Kerr, 2013]

5.2. Arviointi

Seuraavaksi arvioin eri sovelluskehysten yleistä rakennetta, keskeisiä ohjelmistoteknisiä piirteitä sekä niiden vaikutusta sovelluskehyksellä tehtävien sovellusten laatuominaisuuksiin.

5.2.1. Maria Framework

Maria-kehyksen etuja on sen yhtenevyys MVC-mallin alkuperäisen kehittäjän tarkoittaman arkkitehtuurin kanssa. Perinteisen MVC-mallin voidaan olettaa olevan hyvin tunnettu ohjelmistokehityksen alalla. Siksi uusi ohjelmoija tuntenee Maria-kehyksen takana olevat arkkitehtuurilliset periaatteet.

Lähtökohtaisesti Maria-kehyksessä käytetty arkkitehtuuri vähentää oleellisesti sovelluksen sidoksellisuutta. MVC-mallin eri osat on Maria-kehyksen MVC-toteutuksessa erotettu vastuualueiltaan selkeästi toisistaan. Malli, näkymä ja ohjain ovat löyhästi sidottu toisiinsa. Lisäksi hierarkkisen MVC-toteutuksen ansiosta eri osat voidaan tehdä – toteutuskohtaisesta tilanteesta riippuen – toisistaan riippumattomiksi, jolloin sidoksellisuus niin ikään vähenee.

Oleellisen huomion sidoksellisuuden läsnäolosta Maria-kehyksen MVC-mallissa voi kuitenkin tehdä käyttöliittymätoimintojen määrittelemiseen käytetystä toteutuksesta. Käyttöliittymätoimintoja vastaavat funktiot liitetään DOM-elementteihin näkymäluokan puolella [Michaux, 2013], kun alkuperäisen MVC-mallin idean mukaan näkymän ei pitäisi tietää mitään käyttäjäsyötteestä [Reenskaug, 1979]. Tämä on tulosta todennäköisimmin siitä, ettei DOM-elementtejä haluta näkymän asemesta myöskään saattaa ohjaimesta riippuvaiseksi. Toisaalta tässä ratkaisussa on omat hyvät puolensa: ohjaimen ja lähinnä näkymää vastaavien DOM-elementtien välinen sidoksellisuus vähenee, kun näkymä on vastuussa käyttöliittymätapahtumien liittämisestä sopiviin ohjaimen metodeihin.

Lisäksi on otettava huomioon se tosiseikka, että verkkosovellusalusta on luonteeltaan oleellisilta osin erilainen vuoden 1979 MVC-mallin toteutusta

käyttäviin sovelluksiin verrattuna. Sitä paitsi Maria-kehyksen käyttöliittymätapahtumaliitokset on toteutettu siten, että näkymässä vain esitellään luettelo tiettyjen DOM-dokumenttirakenteen elementteihin liittyvien käyttöliittymätapahtumien liittämisestä tiettyihin ohjaimen metodeihin, ja itse käyttöliittymätapahtumien käsittely tapahtuu ohjaimessa. On myös otettava huomioon, että rakenteellisesti verkkoselainpohjainen sovellusarkkitehtuuri asettaa MVC-toteutukselle erilliset vaatimuksensa siitä, kuinka DOM-elementteihin perustuva sovelluksen käyttöliittymän rakenne tulisi liittää näkymän ja ohjaimen väliseen tapahtumankäsittelypohjaiseen toimintalogiikkaan. Tästä huolimatta käyttöliittymätapahtumien käsittelyn alustamista näkymän puolella voitaneen pitää jossain määrin kompromissipohjaisena ratkaisuna.

Laajennettavuus ilmenee Maria-kehyksessä ainakin kolmella tavalla. Ensinnä kehyksen tuki lisäosille lisää kehyksen ja kehyksellä tehtävien ohjelmien joustavuutta. Koska kehyksen toimintaa ja kehyksen nykyisten toimintojen toiminnallisuutta pystytään laajentamaan ja muokkaamaan, Maria-kehyksen päälle tehdyn ohjelman toiminnallisuus ei rajoitu kehyksen tarjoamiin toiminnallisuuksiin. Tämä on entistä tärkeämpää nykyaikaisissa sovelluksissa, koska sovelluksen toimittaminen ei useinkaan ole kertaluontoinen toiminne, vaan sovellus elää sen käyttäjäkunnan tarpeiden mukaan. Lisäosatuen myötä niin ikään sovelluksen ylläpidettävyys paranee. Lisäksi laajennettavuus ilmenee mahdollisuutena muokata kehyksen luokkien toimintaa perinnän avulla. Näin kehyksen toimintaa voidaan mukauttaa sovelluskohtaisten vaatimusten mukaisesti. Lisäosatuen ja perinnän lisäksi laajennettavuutta tukee vielä joukkonäkymien, joukko-ohjainten ja joukkomallien käsitteet. Näkymä voi koostua useasta eri näkymästä ja jopa useasta eri joukkonäkymästä, jotka taas voivat koostua toisista näkymistä tai joukkonäkymistä. Tämä hierarkkisen mallin tuki on huomattava Maria-kehyksen etu, sillä sovelluksesta saadaan modulaarisen toteutustavan ansiosta helpommin ylläpidettävää, kun toteutuskohtaiset yksityiskohdat voidaan sijoittaa alempiin hierarkiatasoihin niin näkymä-, ohjain- kuin mallikohtaisissa toteutuksissa.

Tehokkuuteen Maria-kehyksessä on pyritty panostamaan kehyksen Javascript-tiedostojen minimoinnin ja yhdistämisen avulla; on nopeampi ladata yksi pakattu tiedosto kuin monta pakkaamatonta tiedostoa, useampaa http-pyyntöä käyttäen. Lisäksi tehokkuuteen on panostettu listojen päivittämiseen liittyvän logiikan avulla. Uuden kohteen lisääminen mallijoukkoon aiheuttaa mallijoukon *change*-viestin käynnistymisen. Kun mallijoukosta poistetaan yksittäinen jäsen, lähetetään poistosta kertova *deleted*-tyyppinen viesti

tarkkailijamallia hyödyntäen niin kyseisen jäsenen tarkkailijoille kuin yleisesti koko joukkoa tarkkaileville tarkkailijoille. Koko jäsenjoukkoa tarkkaileville olioille lähetettävässä viestissä ilmoitetaan kuitenkin *deletedTargets*-nimisen parametrin avulla luettelo niistä kohteista jotka poistettiin (muutosten yhteydessä *changedTargets*-viesti). Tällöin tarkkailijat, kuten jäsenjoukkoa kuvaavat näkymät, voivat halutessaan päivittää vain muuttuneet tiedot käyttöliittymään, mikä nopeuttaa laajempien tietojoukkojen päivittämistä käyttäjän selainnäkymään.

Näkymän päivittämisessä tulee vastaan kuitenkin näkymäolion toimintalogiikan osalta tehokkuuteen liittyviä heikkouksia: Näkymän *buildData*-metodi joutuu päivittämään koko listaa vastaavan, DOM-elementteinä esitettävän käyttöliittymässä esiintyvän listauksen, mikä hidastaa käyttöliittymän käytettävyyttä ja suorituskykyä etenkin suurten tietotaulukoiden päivittämisen yhteydessä. Tähän Maria-kehys ei itsessään näytä antavan ratkaisua, joten ainoa tapa listauksissa tapahtuvien päivitysten yksittäisten muutosten aiheuttamia päivityksiä vastaavien DOM-päivitysten suorituskyvyn nostamiseen on toistaiseksi sovelluskehittäjän oma, *buildData*-metodin yhteydessä suoritettava toimintalogiikka, jonka mukaan näkymä päivitetään käsin tarkkailijanotifikaation *deletedTargets*- tai *changedTargets*-viestiä hyödyntäen.

5.2.2. jQueryMX

Kehyksen modulaarinen toteutus mahdollistaa sen, että kehittäjä voi ladata sovelluskohtaisten tarpeiden mukaan vain ne osat, joita tarvitsee sovellustoteutuksessaan. Kehittäjä voi esimerkiksi ottaa käyttöön pelkästään kehyksen MVC-toteutuksen. Toisaalta kehittäjä voi käyttää itsenäisesti kehyksen avustinkirjastoja ilman MVC-toteutuksen lataamista. Näin kehittäjä voi jQueryMX-kehystä käyttäessään minimoida kehystä käyttävän sovelluksen tiedostokoon sekä sivulataukseen liittyvän, tarpeettoman koodikannan aiheuttaman vasteajan, mikä lisää kehyksellä tehtävien sovellusten tehokkuutta. Sovelluskohtaisten tarpeiden mukaisen komponenttiräätälöinnin tukemiseksi jQueryMX-kehyksestä on mahdollista ladata valmis minimoitu versio, joka pohjautuu vain sovelluskohtaisten tarpeiden pohjalta valittuun joukkoon kehyksen eri komponentteja.

JQueryMX-kehyksellä tehdyssä sovelluksessa voidaan katsoa esiintyvän jossain määrin vahvaa sidoksellisuutta. DOM-elementtejä muutetaan suoraan ohjaimesta käsin. Esimerkiksi painikkeen painallukseen liittyvä käyttöliittymätapahtuma liitetään suoraan DOM-rakenteen elementtiin, jolloin uudelleenkäytettävyyden mahdollisuus vähenee. Kun ohjain on vahvasti liitoksissa

sovelluksen käyttöliittymän rakenteeseen, on selvää, että mahdollisia sovelluksen käyttöliittymään kohdistuvia ylläpitotoimia ei välttämättä voida tehdä perehtymättä tarkemmin ohjaimen toteutukseen.

JQueryMX-kehyksessä ei ole erillistä näkymäoliota. Tällöin siirtyy näkymäriippuvaista koodia väistämättä sovelluksen ohjaimen käsiteltäväksi. Näin käyttöliittymäkoodia ei todellisuudessa ole erotettu sovelluksen toiminta-logiikasta, ja sovelluksen siirrettävyys kärsii. Lisäksi näkymän puute haittaa koodin luettavuutta. Ennen tiedon näyttämistä käyttöliittymässä joudutaan tietoa mahdollisesti esikäsittelemään ohjaimen puolella, jolloin ohjaimessa esiintyvän koodin luettavuus kärsii. Kehystä käyttävän kehittäjän on toki mahdollista luoda ohjaimesta erillään oleva funktio tai peräti oma yksinkertainen näkymäluokka, jossa ohjaimen mallilta hakema tieto jalostetaan käyttäjälle näytettävään muotoon. Tällöin kehittäjä ei kuitenkaan saa kehyksen mukana valmiita ratkaisuja niille toimintamalleille, joiden MVC-pohjaisen sovelluskehyksen edellyttäisi tarjoavan.

Joiltain osin jQueryMX-kehyksellä toteutettavissa sovelluksissa voidaan havaita koodin tehokkuuteen ja suorituskykyyn liittyviä puutteita. Jos useasta mallista koostuvassa joukossa tapahtuu muutos, osaa jQueryMX ilmoittaa tarkkailijoille vain niistä riveistä, jotka ovat muuttuneet mallijoukossa. Kuitenkin, koska jQueryMX käyttää näkymän sijasta työpohjia, joudutaan muutoksen tapahtuessa joukon yhdessä oliossa rakentamaan uudestaan koko joukkoa vastaava käyttöliittymäosa, ellei kehittäjä itse toteuta erillistä muuttuneiden tietojen tarkistusta. Koska jQueryMX ei tarjoa älykästä muuttuneiden rivien havainnointitoiminnallisuutta, voidaan tehokkuuden tältä osin päätellä olevan puutteellista jQueryMX-kehyksessä.

Mallien siirrettävyydessä ja yhteentoimivuudessa jQueryMX nousee tietyiltä ominaisuuksilta edukseen. Mallit käyttävät tässä normaalisti tarkkailijamallia, mutta sen lisäksi mallin sisäinen toteutus perustuu niin kutsuttuun *RESTful*-rajapintaan [JavaScriptMVC, 2014], jossa tiedon käsittely ja kuvaus saadaan toteutettua mahdollisimman asiakaspääteriippumattomasti. JQueryMX-kehyksessä käytettävän mallin data voidaan helposti tallentaa kehyksen mukana tulevilla rajapinnoilla esimerkiksi suoraan palvelimelle, jolloin mallista tulee tehokas väline tiedon tallentamiseen palvelimella olevaan tietokantaan. Mallia käyttävä sovellus tai sovelluksen osa ei tiedä mitään tiedon todellisesta tallennuspaikasta, vaan on kiinnostunut vain tiedon hakemisesta kyseiseltä mallilta, riippumatta tiedon todellisesta lopputallennuspaikasta.

JQueryMX-kehyksellä tehdyn sovelluksen laajennettavuus on mahdollista kehyksen tukemien luokkapohjaisten ratkaisujen ansiosta. Perintätuki

mahdollistaa muun muassa olioiden toiminnan laajentamisen ja mukauttamisen. Toisaalta kehyksellä tehtyjä luokkia ei voi suuremmitta ponnisteluitta käyttää uudelleen toisessa sellaisessa projektissa, joka käyttää jotain muuta kuin jQueryMX-kehystä. Kehyksellä on oma syntaksinsa ja toimintamallinsa luokkapohjaisten rakenteiden toteuttamiseen, joten ilman jQueryMX-kehyksen sisällyttämistä projektiin ei voida uudelleenkäyttää kehyksellä tehtyä sovelluskoodia.

Ylläpidettävyyttä ja luettavuutta voi jQueryMX-pohjaisessa sovelluksessa lisätä kehyksen mukana optiona tulevan DocumentJS-kirjaston avulla. Tämän avulla pystytään generoimaan automaattisesti API-dokumentaatio sovelluksen tarjoamien luokkien ja luokkametodien pohjalta. Tällöin muiden samaa sovellusta myöhemmin mahdollisesti muokkaavien sovelluskehittäjien on helpompi päästä sisään sovelluksen sisäiseen arkkitehtuuriin, ja ohjelman luettavuus ja ylläpidettävyys paranee.

Toisaalta jQueryMX-kehys tarjoaa joustavuutta kehyksen yhteydessä käytettävän työpohjakirjaston valinnassa. Kehittäjä voi vapaasti käyttää kolmannen osapuolen jQuery-yhteensopivaa työpohjakirjastoa. Mukana tulee kuitenkin oletuksena muiden muassa jQuery-kirjaston *jQuery.EJS*-työpohjakirjasto [JavaScriptMVC, 2014], joka käyttää Ruby-ohjelmointikielestä tuttua ERB-työpohjastandardia [RubyDoc, 2014].

JQueryMX vaikuttaa jokseenkin suoraviivaiselta kirjastolta Javascript-pohjaisten sovellusten kehittämiseen. Kehyksen avulla useiden verkkosovellus-kehityksessä kohdattavien ongelmien käsittely helpottuu. Muun muassa tapahtumankäsittely, joka on keskeinen osa verkkosovellustenkin kehittämistä, on sovelluksissa yksinkertaista toteuttaa. Kehyksen MVC-toteutus jää kuitenkin vaillinaiseksi, muun muassa todellisen näkymän puutteen vuoksi, sillä näkymän vastuualueisiin liittyviä tehtäviä jää ohjaimen rasitteeksi.

Toisaalta jQuery-laajennuksesta tutun ohjelmointilogiikan hyödyntäminen kehyksessä on iso etu sovelluksen käytön yleistymisen kannalta, sillä juuri jQuery-laajennuksesta on modulaarisine sovellusratkaisuineen kehkeytynyt verkkosovelluskehittäjien käytännön työssä melkein Javascript-kielen jatke.

5.2.3. KnockoutJS

KnockoutJS-kehyksen käyttö vähentää huomattavasti sovelluksen toteut-tamiseen tarvittavaa koodin määrää. Tämä on omiaan nopeuttamaan sovelluskehitystyötä. Tällöin sovelluksen muunneltavuus myöhemmässä vaiheessa saattaa olla helpompaa, kun muuttuja voidaan lisätä tarkkailemaan yhden sijaan useampia elementtejä.

Kehyksen tehokkuus tulee esiin kahtena ominaisuutena. Ensinnä kehyksen tarkkailijamallin toteutuksessa vältellään tarpeetonta viestittelyä. Tarkkailijat eivät tarkkaile koko näkymämallia, vaan vain yksittäistä, kunkin tilanteen kannalta oleellista, näkymämallin attribuuttia. Tämä on selvä etu KnockoutJS-kehyksellä kehitettävässä sovelluksessa, ja se osoittautuu hyödylliseksi etenkin laajempia ohjelmakokonaisuuksia kehitettäessä.

Kehyksen tehokkuus tulee esiin myös DOM-elementtien päivittämiseen liittyvässä älykkyydessä. Kun näkymämallissa muutetaan tarkkailtavaa näkymämallin listaa muuttamalla, poistamalla tai lisäämällä alkio, KnockoutJS-kehys päivittää vastaavaa DOM-näkymää vain muuttuneiden tietojen osalta. Tämä on erittäin tärkeätä suurien tietokokonaisuuksien näyttämisen yhteydessä sekä erityisesti matalan suorituskyvyn mobiililaitteilla.

KnockoutJS-kehys perustuu vahvasti konventiopohjaiseen ratkaisuun; kehittäjä ei itse eksplisiittisesti päivitä kenttien ja muuttujien arvoja, vaan muutokset päivitetään käyttöliittymäkomponenttien ja mallin välillä kehittäjän määrittämien tietoliitosten avulla.

Toisaalta KnockoutJS-kehyksessä on ongelmallista se, että ohjelmalogiikkaan liittyviä määritelmiä joudutaan liittämään HTML-koodin joukkoon. Kun DOM-puuhun liitetään ohjelmalogiikkaan liittyvää koodia, sivuston HTML-rakenne ei ilmennäkään pelkästään sovelluksen näkymän rakennetta, vaan mukana on myös sovelluksen toiminnallisuuteen liittyviä merkintöjä. Tällöin HTML-rakenteen luettavuus voi kärsiä.

Luettavuutta voi heikentää myös se, että kehyksellä tehdyn ohjelman toimintalogiikka on hyvin pitkälle piilotettua. Tämä voi heikentää koodin luettavuutta ja ymmärrettävyyttä etenkin silloin, kun muu sovelluksen koodia muokkaava kehittäjä ei ole perehtynyt riittävällä tasolla kehyksen toimintaan liittyvään ideologiaan. Toisaalta luettavuuden kannalta etuna voidaan pitää nimenomaan sitä, että yksinkertainen ohjelman toimintalogiikka saadaan aikaiseksi hyvin pienellä määrällä koodirivejä. Sitä paitsi sovelluksen kehittäjä voi itse laajentaa kehyksen tukemaa toiminnallisuutta tekemällä omia mukautettuja tietoliitoksia, joiden toimintalogiikka poikkeaa määrätyllä tavalla kehyksen mukana tulevien käyttöliittymätapahtumien edellyttämien vaatimusten viitekehyksestä.

KnockoutJS-kehykseen pohjautuvien sovellusten ohjelmakoodin siirrettävyys toisessa ympäristössä hyödynnettäväksi on heikkoa. Ohjelmassa tehtyjä toimintakokonaisuuksia pystyy siirtämään pienellä vaivalla vain toiseen KnockoutJS-kehystä käyttävään projektiin.

Lisäksi koska HTML-rakenteen sekaan tulee ohjelman toimintalogiikkaan liittyvää koodia – joka pitäisi MVC-mallin perusluonteen mukaisesti olla ohjaimessa – hankaloituu ohjelman osien siirrettävyys ja uudelleenkäyttö, liiallisen koodin sidoksellisuuden vuoksi. Jos HTML-rakennetta halutaan käyttää toisessa projektissa, esimerkiksi erilaisen näkymämallin kanssa, niin joudutaan kaikki ohjelmalogiikka purkamaan HTML-koodin seasta, jotta uudelleenkäyttö olisi mahdollista.

Myöskään toiseen, lähempänä perinteistä MVC-pohjaista ratkaisua käyttävän kehyksen mukaiseen sovellukseen ei ole helppo siirtää KnockoutJS-kehykseen pohjautuvaa koodia, sillä KnockoutJS-kehyksen tietoliitokset ovat käsitteellisesti kaukana perinteisen MVC-mallin yhteydessä käytetystä tarkkailijamallista. Jos koodia ei haluta tai ei ole tarve hyödyntää kehyksettömässä tai toisella kehyksellä tehdyssä koodissa, osoittautuu tietoliitosten käyttö erittäin näppäräksi perinteisten tarkkailijamallipohjaisten toteutusten sijasta käytettäväksi ratkaisuksi.

Kehyksellä toteutettavassa ohjelmakoodissa voidaan kuitenkin havaita esiintyvän näkymämallin käytöstä aiheutuvaa korkeaa sidoksellisuutta. Laajempia muutoksia tekeviä käyttöliittymätapahtumia vastaavat funktiot nimittäin sijoitetaan suoraan näkymämalliin, jolloin ohjelman tieto ja sen käsittely suoritetaan samassa luokassa. Tällöin myös näkymämallin koodi saattaa paisua liian suureksi, jolloin ohjelman toimintalogiikan uudelleen-käytettävyys voi kärsiä. Myös koodin luettavuuteen saattaa tiedon haun ja käsittelyn yhdistämisellä olla haittavaikutuksia, etenkin laajemmissa ohjelmistokokonaisuuksissa.

Luettavuutta voi haitata myös se, että samaan luokkaan, näkymämalliin, sijoitetaan sekä näkymään että malliin liittyviä toimintakokonaisuuksia. Tämän vuoksi mallista haettava teksti, vaikka käyttäjän lokaaliasetusten mukainen rahasumma, joudutaan muotoilemaan sovelluksen käyttäjälle näytettävään muotoon nimenomaan näkymämallissa, ei erillisessä näkymäoliossa. Tästä aiheutuu se, että ohjelmakoodia voi vähitellen olla vaikea lukea ja ymmärtää: liikaa eri vastuualueisiin liittyvää koodia yhdistettynä samassa paikassa suoritettavaksi heikentää koodin luettavuutta.

KnockoutJS-kehyksen tietoliitosten ansiosta sovelluksessa käyttäjälle esitettävät tiedot pysyvät synkronoituna automaattisesti. Käyttöliittymää ei näin ollen tarvitse manuaalisesti päivittää vastaamaan malliolioiden muuttujien kulloistakin tilaa. Kehyksen tietoliitosten paradigma muistuttaa jokapäiväisen tietojenkäsittelyn näkökulmasta lähinnä taulukkolaskentasovellusten solu-

viittausten ja -funktioiden käyttäytymistä. Solun arvon muuttaminen saa aikaan soluun viitanneiden solujen kaavojen uudelleenlaskun.

KnockoutJS-kehys käyttää deklaratiivisia tietoliitoksia. Tämä on vastakohta vallitsevalle imperatiiviselle ohjelmointiparadigmalle, minkä ansiosta sovelluksen kehittäminen voi nopeutua. Kehyksessä käytetyn tietoliitosmekanismin ansiosta myöskään erilliselle ohjaimelle ei välttämättä ole perusteltua tarvetta.

Tietoliitospohjainen ohjelmointiparadigma pienentää myös kehyksellä tehtävän sovelluksen kokoa. Koska KnockoutJS-kehyksellä tehdyn sovelluksen toimintalogiikka perustuu pitkälti kehyksen sisäisen ohjelmalogiikan toistamiseen, tulee kehyksellä tehdyksi ohjelmakoodiltaan suhteellisen pienikokoisia ohjelmia.

Myös sovelluksen kehittäminen KnockoutJS-kehyksellä on suhteellisen nopeaa. Tähän vaikuttaa osaltaan se, että ei tarvitse yksilöidä DOM-elementtejä nimeämällä niitä, sillä mallin muuttujiin sekä tavallisesti ohjaimessa esitettäviin käyttöliittymätapahtumiin viitataan suoraan HTML-koodista. Ei tarvita ohjainta, jossa käyttöliittymätapahtumia valvottaisiin ja niiden mukaan vastaavasti mallin sisältöä muutettaisiin.

Niin ikään sovelluksen ylläpito helpottuu, kun sovellus tehdään KnockoutJS-kehyksen pohjalle. Erityisesti ylläpidettävyyttä helpottaa se, että lisätessään myöhemmin uusia toiminnallisuuksia sovellukseen kehittäjän ei tarvitse huolehtia, mitä DOM-elementtejä tulisi muutosta vastaavalla tavalla päivittää: kehys huolehtii DOM-elementtien päivityksen, kun näkymämallin attribuutteja lisätään, poistetaan tai muutetaan. Myös dokumentaation laajuus ja kypsyys on ylläpidettävyyden kannalta selvä etu KnockoutJS-kehyksessä. KnockoutJS-kehyksen sivulla on dokumentaation ja ohjelmointirajapintakuvausten lisäksi runsaasti tarjolla koodiesimerkkejä sekä vaiheittain eteneviä tutoriaaleja. Esimerkit lähestyvät yleisiä verkkosovellusten kehityksessä mahdollisesti kohdattavia ongelmia vaihe vaiheelta edeten, joten myös uusien projektiin myöhemmin mukaan tulevien sovelluskehittäjien työ helpottuu.

Kehyksessä sovellettu ohjelmointiparadigma helpottaa sovelluksen päivitettävyyttä ja sitä kautta niin ikään ohjelman ylläpidettävyyttä. Perinteisiin ratkaisuihin nojautuvien, monimutkaisten sovellusten jatkokehittelyn yhteydessä voi ilmetä vaikeuksia muistaa, mihin eri elementteihin yhteen muuttujaan tehty muutos tulee päivittää. Kun taas käytössä ovat tietoliitokset, kehyksen koodi hoitaa tarvittavien komponenttien tilan tai arvon päivityksen automaattisesti. Toisaalta kehittäjälle saattaa aiheutua ylimääräistä työtä tapauksissa, joissa tehtyjä muutoksia ei halutakaan välittömästi välittää näkymälle.

5.3. Vertailu

Jaottelu kriteereittäin mahdollistaa järjestelmällisen tavan kehysten vertailuun. Vertailussa tarkastellaan sovellusten sidoksellisuutta, luettavuutta, tehokkuutta, siirrettävyyttä, ylläpidettävyyttä sekä ohjelmakoodin kokoa. Lopuksi esitetään vertailun tulokset ja johtopäätökset.

5.3.1. Sidoksellisuus

Arvioitujen sovelluskehysten sidoksellisuus kiteytyy vahvasti MVC-mallin sekä sen komponenttien välisen viestinnän toteutustapaan. Maria-kehyksessä malli, ohjain ja näkymä on selkeästi eritelty omiin luokkiinsa, kun taas jQueryMX-kehyksessä oman luokkansa saavat vain ohjain ja malli. Toisaalta KnockoutJS-kehyksessä näkymän ja mallin tehtävää hoitaa näkymämalli, ja ohjainta ei määritellä ollenkaan kehyksessä käytettävien tietoliitosten toimintatavan vuoksi.

Kaikki kehykset pyrkivät vähentämään toteutettavan sovelluksen osien välistä sidoksellisuutta, käyttämällä muun muassa mallin ja näkymän välisessä tiedonsiirrossa tarkkailijamallia tai sen johdannaista. Tällöin mallin ei tarvitse tietää, mitä näkymät tekevät, kun ne päivittävät tilansa mallin uutta tilaa vastaavaksi. Maria-kehyksessä osien väliseen riippumattomuuteen pyritään perinteistä tarkkailijamallia noudattaen, kun taas KnockoutJS-kehyksessä tarkkailijamallin toiminta on sisällytettynä tietoliitoksen logiikkaan.

Vähäisen sidoksellisuuden ansiosta sovelluksen yhtä osaa, esimerkiksi ohjainta, on helpompi käyttää toisessa sovelluksessa, riippumatta muiden osien tarkasta toteutuksesta. Lisäksi eri sidosryhmät voivat mukauttaa sovelluksen ulkoasua, mallin sisäistä tiedonkäsittelyä sekä ohjaimen toimintaa itsenäisemmin: vähäiset muutokset koodin eri osissa eivät välttämättä vaadi muutoksia muissa ohjelman osissa.

Mikään vertailluista kehyksistä ei pyri toteutuksellaan täydelliseen riippumattomuuteen MVC-osien välillä. Tietyiltä osin tämä on täysin ymmärrettävää. Onhan esimerkiksi näkymän tiedettävä sen mallin rakenne, jota näkymä graafisesti yrittää kuvata. Lisäksi on luonnollista, että ohjaimen täytyy olla tietoinen rajapinnasta, jolla mallin sisältämää dataa muutetaan.

Sidoksellisuuden tarkastelun kannalta on myös tärkeää huomioida, kuinka käyttöliittymätapahtumat liitetään ohjaimen tai toteutuksen mukaisen, ohjainta vastaavan komponentin käyttöliittymätapahtumafunktioihin. Esimerkiksi Maria-kehyksessä DOM-elementtien käyttöliittymätapahtumat liitetään ohjaimen funktioihin näkymän puolella. Perinteisen MVC-mallin mukaisesti ohjainten käsittelyn tulisi sisältyä kokonaan ohjaimen vastuualueeseen, jopa

niin että näkymän tulisi olla mahdollista olla olemassa jopa ilman ohjaimen näkymään liittämistä.

JQueryMX-kehyksessä käyttöliittymätapahtumat liitetään ohjaimen funktioihin kokonaan ohjaimen puolella, jolloin ohjain on riippuvainen näkymän DOM-rakenteesta. Tähän saattaa osaltaan olla vaikuttamassa se, ettei kehyksen toteutuksessa ole varsinaista näkymäoliota. Tämän seurauksena sovelluksen käyttöliittymä on vahvasti sidoksissa sovelluksen ohjaimeen.

Maria- ja jQueryMX-kehyksistä poiketen KnockoutJS lähestyy käyttöliittymätapahtumien liittämisen ongelmaa täysin toisesta näkökulmasta. Tapahtumien liittäminen tiettyihin DOM-elementteihin määritellään suoraan sovelluksen DOM-rakenteessa HTML-elementtien *data-bind*-määrettä käyttäen. Tällöin sovelluksen käyttöliittymään liittyvä rakenne ja käyttöliittymätapahtumien määrittely sijaitsevat samassa paikassa, mikä nopeuttaa sovelluksen toteuttamista. Toisaalta tässäkin lähestymistavassa sovelluksen rakenteen ja käyttöliittymän ohjauksen välinen sidoksellisuus korostuu.

5.3.2. Luettavuus

Koodin luettavuus on pitkälti subjektiivinen kokemus, sillä luettavuuteen ja ymmärrettävyyteen vaikuttaa yleisten luettavuutta kohentavien ohjelmointitapojen lisäksi koodia tarkastelevan sovelluskehittäjän tausta.

Yleisesti ottaen Maria-kehyksen luettavuus on parhaimmillaan sellaisten sovelluskehittäjien kohdalla, jotka haluavat eksplisiittisemmin nähdä, mitä sovellus tekee missäkin kohtaa koodia. Kehyksellä toteutetun sovelluksen mallin saantifunktioissa esimerkiksi erikseen ilmoitetaan tarkkailijoille mallin tilan muuttumisesta, kun taas KnockoutJS-kehyksessä mallin tilan muuttumisesta ilmoittaminen on automatisoitu. Mallin tilan muuttumisesta ilmoitetaan automaattisesti mallin attribuuttia tarkkaileville tarkkailijoille, eli tässä tapauksessa suoraan niille käyttöliittymän HTML-elementeille, joissa mallin attribuutti on mainittu osana elementin arvon määritelmää.

Jotkut sovelluskehittäjät saattavat kokea ongelmalliseksi sen, että KnockoutJS-kehyksessä näkymä ja malli ovat samassa luokassa. Luettavuuteen tämä saattaa vaikuttaa alentavasti esimerkiksi sellaisessa tapauksessa, jossa näkymämallin toimintalogiikka laajenee tavallisen sovelluskehittäjän hallitsemaa koodikokonaisuutta laajemmaksi.

Toisaalta jQueryMX-kehyksessä luettavuutta voi hankaloittaa – etenkin laajempia ohjelmakokonaisuuksia ohjelmoitaessa – erillisen näkymäolion puute. Kun ohjaimeen alkaa ilmestyä yhä enemmän näkymään liittyvää koodia, sovelluskoodin luettavuus ja niin ikään ylläpidettävyys vaikeutuvat.

5.3.3. Tehokkuus

Tehokkuuteen liittyvät huomiot koskevat esitellyissä sovelluskehyksissä erityisesti tarkkailijamallin toteutuksen suorituskykyä. Lisäksi tehokkuuteen vaikuttaa sovelluskehyksen Javascript-kirjaston koko sekä kehyksessä käytetty ohjelmointiparadigma.

Esitellyistä sovelluskehyksistä jokainen oli toteutettu siten, että sovelluskehyksestä on tarjolla sekä pakkaamaton että pakattu versio. Näiden sovelluskehysten tapauksessa suorituskykyä on pyritty lisäämään minimoimalla kehysten ohjelmakoodi sekä liittämällä eri luokkia vastaavat tiedostot yhteen tiedostoon. Tämä toimintatapa oli esillä kaikissa kehyksissä. Tämän lisäksi kuitenkin jQueryMX-kehys on ottanut huomioon myös sovelluskehittäjien poikkeavat sovelluskehystarpeet, mahdollistamalla räätälöidyn komponenttikokoelman sisältävän sovelluskehyskirjaston rakentamisen. Tällöin sovelluksen lataamiseen kuluvaa aikaa saadaan entisestään vähennettyä, kun tarvitsee ladata vain kehitettävän sovelluksen tarpeiden mukaiset sovelluskehyksen osat.

Sovelluskehyksillä tehtävän sovelluksen suorituskykyyn vaikuttaa oleellisesti myös viestittelyn määrä. Vertailluista sovelluskehyksistä kaikki toteuttivat tarkkailijamallin siten, että vain muuttuneista riveistä ilmoitetaan tarkkailijoille. Näistä poiketen kuitenkin KnockoutJS-kehys on vienyt tarkkailijamallin toteutuksen vielä pidemmälle. Siinä tehokkuutta on lisätty siten, että tarkkailijat tarkkailevat vain yksittäistä näkymämallin attribuuttia, eivät koko näkymämallia. Lisäksi tehokkuus ilmenee KnockoutJS-kehyksessä siten, että tarkkailijailmoituksen tapahtuessa kehys päivittää – muista kehyksistä poiketen – automaattisesti vain ne osat käyttöliittymästä, jotka ovat muuttuneet. Jos listasta esimerkiksi poistetaan yksi rivi, vain kyseinen rivi poistetaan näkymästä, eikä koko listaa rakenneta näkymään uudestaan.

5.3.4. Siirrettävyys

Maria-kehyksessä malli, ohjain ja näkymä ovat melko riippumattomia toisistaan, joten myös niiden erillinen uudelleenkäyttö toisessa, samaa kehystä käyttävässä projektissa on helppoa. Toisaalta jQueryMX-kehyksessä on erityisesti huomioitu kehyksellä rakennettavan sovelluksen mallin siirrettävyyden mahdollistaminen. Kehyksen mallin käyttämä RESTful-rajapinta mahdollistaa mallin datan tallentamisen mihin tahansa tämän mallin rajapintaa ymmärtävän sovelluksen tietokantaan tai muuhun lopputallennus-paikkaan. Muissa kehyksissä vastaava toiminnallisuus on itse ohjelmoitava, sovelluskohtaisten lähtökohtien pohjalta.

Kaikissa tapauksissa siirrettävyyttä heikentää oman, sovelluskehys-kohtaisen luokkapohjaisen rakenteen toteuttaminen. Tietyn kehyksen piirissä kehitettyjen luokkien uudelleenkäyttö muita kehyksiä käyttävissä sovelluksissa on hankalaa, sillä sovellus toteutetaan kehyksen määrittelemän olio-mallinnuksen lainalaisuuksia noudattaen. Tämän lisäksi siirrettävyyttä hankaloittaa kunkin kehyksen omat MVC-toteutustavat sekä kehyskohtaisten funktioiden käyttö.

Tietyn kehyksen pohjalle rakennetun sovelluksen yksittäisen luokan toiminnallisuuksien käyttö toisessa projektissa on kuitenkin mahdollista, mikäli sovelluskehittäjä on valmis käyttämään aikaa sovelluksen toimintojen ja funktioiden mukauttamiseen toiseen ympäristöön. Kuitenkin jo sovelluksen päivittäminen saman sovelluskehyksen uudempaan versioon tuottaa usein hankaluutta, joten tietyissä tapauksissa voi saman toiminnallisuuden kirjoittaminen kokonaan uudelleen olla järkevämpi ratkaisu.

Toisaalta on ymmärrettävää, että siirrettävyys hankaloituu, kun päädytään käyttämään sovelluskehystä sovelluksen toteutukseen. Koska sovellus-kehyksen perimmäinen idea on tarjota valmiita ratkaisumalleja sovelluksen toiminnallisuuksien toteuttamiseen, on luonnollista, että sovelluksen uudelleenkäyttö toista kehystä käyttävässä sovelluksessa hankaloituu.

Joka tapauksessa koko sovelluksen toteutuksen kannalta on siirrettävyyden taso kaikissa vertailluissa kehyksissä riittävä, sillä käytettävä kehys ei ratkaisevasti vaikuta siihen, missä ajoympäristössä sovellusta voidaan ajaa. Selainpohjaisen verkkosovelluksen luonteen vuoksi tarvitaan vain selain, joka on Javascript- ja HTML-kielten standardien mukainen. Tässä mielessä siirrettävyys on monesta näkökulmasta tarkasteltuna automatisoidumpaa kuin esimerkiksi työpöytäsovelluksissa.

5.3.5. Ylläpidettävyys

Sovelluksen ylläpidettävyyteen vaikuttaa omalta osaltaan sovelluskehyksen määrittelemä sovellusarkkitehtuuri. Vertailluista kehyksistä Maria-kehys mahdollistaa ylläpidettävyyden muun muassa lisäosatuen avulla. Sovelluksen käytön jatkumisen kannalta on tietyissä tapauksissa oleellista, että sovelluksen toimintaa voidaan mukauttaa lisälaajennusten avulla.

Ylläpidettävyyden kannalta on myös tärkeää sovelluksen rakenteen ymmärtäminen. Usein sovelluksen käyttöelinkaaren aikana sovellusta ylläpitävät useammat eri sovelluskehittäjät. Tämän takia on keskeistä, että sovelluksen rakenne saadaan selkeästi kuvattua API-dokumentaation avulla. Sovelluksen omaan rakenteeseen liittyvän dokumentaatio-ongelman ratkaisemiseen on muista kehyksistä poiketen kiinnitetty huomiota erityisesti jQueryMX-kehyksessä. Sen mukana tulee valmis, kehyksellä toteutettavien sovellusten arkkitehtuuria ymmärtävä, API-dokumentaation automaattisen generoinnin mahdollistava työkalu. Tämän avulla myöhemmin sovelluskehystä ylläpitävät – eli niin ikään muokkaavat – sovelluskehittäjät voivat perehtyä sovelluksen sisäiseen arkkitehtuuriin.

Lisäksi ylläpidettävyyden mahdollistumiseen vaikuttaa välillisesti myös sovelluskehyksen oman dokumentaation kattavuus. Kaikkien sovelluskehysten osalta dokumentaatio on melko kattava, mutta erityisesti KnockoutJS-kehys on panostanut dokumentaation laajuuteen ja kypsyyteen. Kehyksen verkkosivuilla on dokumentaation ja API-rajapinnan kuvauksen lisäksi sovelluskehyksen keskeisten toimintojen hyödyntämisessä avuksi olevien toiminnallisuuksien havainnollistamiseen keskittyviä, vaihe vaiheelta eteneviä koodiesimerkkejä ja -kuvauksia.

5.3.6. Laajennettavuus

Laajennettavuus on ominaisuutena jossain määrin lähellä ylläpidettävyyttä. Jos sovelluksen laajennettavuus voidaan kokea riittäväksi, myös sovelluksen ylläpidettävyys on helpompaa, sillä usein sovelluksen ylläpitoon liittyy keskeisesti olemassa olevien toimintojen muokkaaminen sekä laajentaminen.

Maria-kehyksessä laajennettavuuteen vaikuttaa myönteisesti erityisesti se, että näkymän on mahdollista koostua useasta eri alinäkymästä. Tämän ansiosta sovelluksen käyttöliittymää voidaan laajentaa, koskematta käyttöliittymän muihin osiin. Lisäksi laajennettavuutta edistää edellä mainittu lisäosatuki, jonka avulla sovelluskehyksen toiminnallisuuden laajentamista saadaan jossain määrin helpotettua.

Myös kehyksen ja sovelluksen luokkien perintätuki mahdollistaa sovelluksen ja kehyksen laajennettavuutta edistävien ohjelmointikäytäntöjen toteuttamista. Tämä oli vertailluista kehyksistä mahdollista Maria- ja jQueryMX-kehyksissä. Lisäksi KnockoutJS-kehyksissä kehyksen omien toimintojen laajentaminen ja mukauttaminen on mahdollista; tämä saadaan aikaan mukautettujen tietoliitosten avulla, mitä ei tosin voitane pitää laajennettavuuden kannalta niin etuna kuin heikkoutena muihin kehyksiin verrattuna.

5.3.7. Koko

Sovelluksen koon määrittäminen on usein helppoa, sillä se voidaan mitata kvantitatiivisesti. Sovelluksen koko riippuu sovelluksen laajuudesta. Siksi tässä määritellään sovelluksen kokonaiskoon lisäksi käytetyn sovelluskehyksen tuoman lisäkoon määrää (ks. taulukko 2).

Kehys	**Versio**	**Koko**	**Koko pakattuna**
Maria	1.1.0	19 kt	10 kt
jQueryMX	1.3.2	12 kt	7 kt
KnockoutJS	3.1.0	46 kt	17 kt

Taulukko 2. Eri kehysten kokovaatimukset.

Koot on laskettu kehyksen *.js*-lähdekooditiedoston tiedostokoosta minimoidussa komentosarjamuodossa. Koko pakattuna ilmaisee, kuinka paljon dataa joudutaan siirtämään, kun tiedosto lähetetään palvelimelta asiakkaan selaimeen http-yhteyden pakkausta käytettäessä. Käytännössä vertailtava koko on juuri pakattu koko, sillä useissa palvelimissa yhteydet palvelimelta selaimelle pakataan oletuksena.

Mitä enemmän sovelluskehyksen koodi vie tilaa, sitä hitaammin sovellus saattaa latautua käyttäjän selaimeen. Luonnollisesti myös muut tekijät vaikuttavat, kuten verkkoyhteyden nopeus, käytetty selain, käyttöjärjestelmä ja laitteisto.

Vertailluilla kehyksillä on käytännössä hyvin pieniä kokoeroja. Koska nykyiset verkkoyhteysnopeudet ovat kasvaneet nopeasti, eivät näin pienet tiedostokoot enää hidasta sivuston latautumista juuri ollenkaan. Koon merkitystä vähentää myös selainten välimuistin käyttö. Koska selaimet säilyttävät ladatut tiedostot välimuistissa, kehyksen komentosarjatiedoston lataamisesta aiheutuva pieni viive kohdistuu vain sivuston ensimmäiseen latauskertaan.

Vastoin mitattavan ohjelmakoon tarkkailua voidaan kuitenkin tehdä yleisiä huomioita sovelluskehyksillä toteutettavan koodin laajuudesta. Yleisesti voidaan huomata, että erityisesti KnockoutJS-kehyksellä toteutetun sovelluksen koodi on verrattain melko kompaktia, sillä huomattava osa sovelluksen toiminnallisuudesta tapahtuu kehyksen sisäistä toimintalogiikkaa hyödyntäen. On pyritty välttämään ylimääräisiä luokkarakenteita sovelluksen arkkitehtuurin määrittelyssä. Tapahtumat määritellään suoraan HTML-rakenteessa, muuttuja-arvojen liittäminen näkymämallin ja HTML-elementtien välillä ilmaistaan suoraan HTML-rakenteessa, ja tietoliitoksien toteuttamiseen käytetään sovelluskehyksen sisäistä ohjelmakoodia, jolloin yleisimmät sovelluksen kehittämiseen tarvittavista toimintamalleista saadaan toteutettua valmiina kehyksen mukana tulevia ratkaisuja hyödyntäen. Nämä KnockoutJS-kehyksen ominaispiirteet vähentävät omalta osaltaan toteutettavan sovelluksen ohjelmakoodin kokoa.

Muut vertaillut sovelluskehykset, Maria ja jQueryMX, taas käyttävät KnockoutJS-kehyksestä poiketen imperatiivista ohjelmointiparadigmaa, jolloin ohjelmakoodissa tulee väistämättä ilmaistua ohjelman toimintalogiikkaa eksplisiittisemmin ja yksityiskohtaisemmin. Tämän myötä myös toteutettavan sovelluksen ohjelmakoodi voi olla monisanaisempaa kuin KnockoutJS-kehyksellä tehdyssä sovelluksessa.

5.3.8. Vertailun yhteenveto

Vertailun yhteenvetona voidaan todeta, että kehyksen arkkitehtuurista aiheutuvissa laatuominaisuuksissa ilmenee osittain yhtäläisyyksiä kuin myös selkeitä arkkitehtuurillisia koodin laatuun vaikuttavia eriäviä tekijöitä. Nämä laatuun vaikuttavat tekijät on ilmaistuna taulukossa 3.

Ominaisuus	Maria	jQueryMX	KnockoutJS
Yleistä	• Alkuperäinen MVC-malli	• MVC-malli ilman näkymää • jQuery-kirjastosta tuttu ohjelmointilogiikka	• MVVM-malli • Käytössä ovat tietoliitokset • Ohjaimelle ei tarvetta
Sidoksellisuus	+ Malli, näkymä ja ohjain ovat luokkia + Näkymä voi koostua useasta alinäkymästä – Tapahtumat liitetään näkymässä	– DOM-elementtejä muokataan suoraan ohjaimesta käsin – Ei ole erillistä näkymäoliota	– Ohjelmalogiikkaan liittyvää koodia HTML-rakenteen seassa – Näkymämallissa niin sovelluksen data kuin tapahtumafunktiot
Luettavuus	+ Näkymän, mallin ja ohjaimen toimintalogiikat riittävän hyvin eriytetty toisistaan	+ Sovelluksen API-dokumentaation automaattinen generointi + Kehittäjät tuntevat hyvin jQuery-pohjaisen tapahtumankäsittelylogiikan – Ei erillistä näkymäoliota	–/+ Ohjelman toimintalogiikka pitkälle piilossa – Ohjelmalogiikkaan liittyvää koodia HTML-rakenteen seassa – Näkymä ja malli samassa
Tehokkuus	+ Javascript-kirjastot minimoituja ja yhdistettyjä + Tarkkailijamallin toteutus: informoidaan vain joukon muuttuneista riveistä – Oletuksena koko näkymäosa päivitetään uudestaan muutoksen tapahduttua	+ Javascript-kirjastot minimoituja ja yhdistettyjä + Tarkkailijamallin toteutus: informoidaan vain joukon muuttuneista riveistä + Vain sovelluksen tarvitsemat osat voidaan ladata – Koko näkymäosa päivitetään uudestaan muutoksen tapahduttua	+ Javascript-kirjastot minimoituja ja yhdistettyjä + Koodi tiivistä + Yksittäistä attribuuttia tarkkaillaan, ei koko näkymämallia + Näkymästä päivitetään vain muuttunut osa + Deklaratiivinen ohjelmointitapa + Data ja HTML-rakenne synkronoituja
Siirrettävyys	– Luokkia ei voi uudelleen käyttää muita kehyksiä käyttävissä sovelluksissa	– Luokkia ei voi uudelleen käyttää muita kehyksiä käyttävissä sovelluksissa – Ei erillistä näkymäoliota + Mallissa RESTful-toteutus	– Sovelluskoodia ei voi hyödyntää muita kehyksiä käyttävissä sovelluksissa
Ylläpidettävyys	+ Lisäosatuki + Tuki joukkonäkymille, -ohjaimille ja -malleille	+ Sovelluksen API-dokumentaation automaattinen generointi	+ Vain muuttuneet rivit päivitetään näkymässä + Kypsä dokumentaatio
Laajennettavuus	+ Tuki alinäkymille + Lisäosatuki + Kehys- ja sovellus-luokkien perintä + Tuki joukkonäkymille, -ohjaimille ja -malleille	+ Sovellusluokkien perintä + Näkymän työpohjakirjaston voi vapaasti valita – Elementtejä muokataan suoraan ohjaimesta	+ Kehittäjä voi itse luoda mukautettuja tietoliitoksia
Koko	– Imperatiivinen ohjelmointitapa	– Imperatiivinen ohjelmointitapa	+ Ohjelmakoodi kompaktia (tietoliitostenkin vuoksi)

Taulukko 3. Eri kehysten laatuominaisuuksiin vaikuttavat tekijät.

Jokaisessa kehyksessä tehokkuuteen on panostettu kuta kuinkin yhtäläisin menettelytavoin, joskin eritoten KnockoutJS-kehyksessä tehokkuus korostuu näkymän päivittämiseen liittyvässä älykkyydessä. Lisäksi siirrettävyyteen vaikutti kaikissa kehyksissä se, että kehyksellä tehtyjen sovellusten luokkarakenne on riippuvainen kehyksen asettamasta luokkarakennemääritelmästä.

Luettavuuteen on kiinnitetty huomiota jokaisessa kehyksessä, joskin luettavuutta on kohennettu eri kehyksissä erilaisista näkökulmista käsin: Maria-kehystä käyttävän ohjelman koodi on eksplisiittistä ilmaisua korostavien sovelluskehittäjien näkökulmasta tarkasteltuna hyvin luettavaa, kun taas KnockoutJS-kehyksen luettavuus ilmenee deklaratiivisen ohjelmointitavan ytimekkyytenä. Toisaalta jQueryMX-koodin luettavuutta parantaa tuttuus jQuery-kirjaston menettelytapojen kanssa.

Sidoksellisuuden osalta voidaan todeta, että Maria-kehyksen selkeä erottelu mallin, näkymän ja ohjaimen tehtävien välillä vähentää Maria-kehyksellä tehtävien sovellusten sidoksellisuutta. Toisaalta KnockoutJS- ja jQueryMX-kehyksissä ohjelman data ja käyttöliittymä ovat vahvasti sidoksissa toisiinsa. Laajennettavuuteen oli kaikissa kehyksissä kiinnitetty huomiota jollain tasolla. Erityisesti Maria- ja jQueryMX-kehyksissä luokkia voitiin laajentaa perinnän avulla. Lisäksi KnockoutJS-kehyksessä kehittäjä pystyy itse määrittelemään toteutettavan sovelluksen kannalta keskeisiä mukautettuja tietoliitossääntöjä.

Lopulta keskeinen huomio sovelluskehysten välillä liittyy yleisen ohjelmistoarkkitehtuurin vaikutuksesta kehyksellä toteutettavan sovelluksen ohjelmakokoon. Maria- ja jQueryMX-kehysten imperatiivinen ohjelmointitapa lisää koodirivejä monisanaisemman ilmaisutavan vuoksi, kun taas KnockoutJS-kehyksen tietoliitospohjainen deklaratiivinen ohjelmointitapa johtaa useissa tapauksissa kompaktimpaan ohjelmakoodiin.

5.4. Vertailun lopputulos

Kaikki läpikäydyt sovelluskehykset lisäävät ohjelmakoodin laatua jossain määrin. Muun muassa ohjelmakoodista voi kehyksen avulla saada vähemmän sidoksellista. Kehykset vähentävät myös oleellisesti usein toistuvan ohjelmakoodin esiintyvyyden tarvetta. Myös tiettyihin tehtäviin, kuten yksinkertaistettuun DOM-hakuun, liittyvät koodiosuudet on otettu kehysten arkkitehtuurissa huomioon.

Arvioinnin ja vertailun pohjalta havaitaan, että eri laatuominaisuuksien vallitsevuus on sidoksissa yhteen tai useampaan toiseen laatuominaisuuteen. Esimerkiksi ohjelman siirrettävyyteen sekä ylläpidettävyyteen vaikuttaa ohjelmakoodin sidoksellisuus, ja toisaalta sidoksellisuus voi vaikuttaa

heikentävästi koodin luettavuuteen. Täten eri laatuominaisuuksien arviointia ei koskaan voida toteuttaa irrallisina kokonaisuuksina, vaan huomioon on aina otettava se, miten eri laatuominaisuudet vaikuttavat muihin laatuominaisuuksiin.

On myös havaittavissa, että toisen laatuominaisuuden läsnäolo saattaa vaikuttaa negatiivisesti johonkin toiseen laatuominaisuuteen. Esimerkiksi äärimmäisen sidoksellisuuden vähyyden tavoittelu saattaa johtaa sovelluksen heikentyneeseen ylläpidettävyyteen. Tämän vuoksi kehystä valitessaan sovelluskehittäjän tulisi ehkä pitää silmällä, mitkä laatuominaisuudet ovat toteutettavan sovelluksen asettamien arkkitehtuurivaatimusten kannalta keskeisiä.

Edelleen voidaan arvioitujen sovelluskehysten pohjalta havaita yleisesti vallitseva MVC-mallin monitulkintaisuus. Arvioiduista kehyksistä jQueryMX-kirjastossa oli jossain määrin havaittavissa puutteita todellisen MVC-arkkitehtuurin toteutuksessa, sillä esimerkiksi näkymälle ei ollut määritelty omaa erillistä komponenttiaan. Tästä voidaan päätellä, että suunnitteilla olevalle verkkosovellukselle sopivaa sovelluskehystä etsivän sovelluskehittäjän suositeltaisiin perehtyvän tarkasti muun muassa sovelluskehyksen dokumentaation koodiesimerkeissä esitettyyn MVC-arkkitehtuurin toteutustapaan sekä muiden osapuolien tekemiin arkkitehtuuriin keskittyviin arvioihin. Sovelluskehyksen oman verkkosivuston esittely ei näin ollen pelkiltään anna välttämättä oikeaa kuvaa todellisesta MVC-mallin toteutuksesta kyseisessä sovelluskehyksessä.

Tutkielmassa vertailluista kehyksistä voidaan Maria-kehystä pitää arkkitehtuurillisesti konservatiivisimpana. Alkuperäisen MVC-mallin kanssa yhtenevä toteutus helpottaa koodin ymmärtämistä, mutta samalla se voi aiheuttaa paljon manuaalista työtä ohjelmoijalle. Koska tietoliitoksia ei voida käyttää, joutuu näkymäolio päivittämään malliin kohdistuneet muutokset imperatiivista ohjelmointiparadigmaa muistuttavalla tavalla, erikseen sovelluksen HTML-rakennetta muuttamalla. Kehyksen arkkitehtuurin pohjalta voidaan päätellä, että Maria-kehys sopii parhaiten sellaisille ohjelmoijille, jotka haluavat verkkosovelluksensa pohjalle selkeästi eri toimialueet eriyttävän sovelluskehyksen. Lisäksi Maria-kehys sopii kehittäjille, jotka haluavat tarkemmin nähdä sovelluksen toiminnallisuuteen liittyvän ohjelmalogiikan.

Toisaalta sovelluksen kehitystyö nopeutuu, kun käytetään KnockoutJS-kehystä. Valmis sovellus saadaan nopeammin loppukäyttäjien ulottuville. Niin ikään tietoliitosten avulla tiedon siirtämistä näkymän ja mallin välillä ei jouduta tekemään manuaalisesti. Tämän ansiosta myös varhaisia testiversioita

saadaan nopeasti testikäyttäjien käytettäväksi. Voidaan päätellä, että KnockoutJS-kehys sopii arkkitehtuuriltaan parhaiten nykyaikaisten ketteriä ohjelmointimenetelmiä suosivien pienten ja keskisuurten yritysten käyttötarpeisiin.

JQueryMX taas tarjoaa joukon hyödyllisiä verkkosovelluskehityksessä tarvittavia komponentteja, jotka ovat avuksi verkkopohjaisten sovellusten kehittämisessä. Sovelluksen laajennettavuuden ja ylläpidettävyyden kannalta voidaan kuitenkin todeta, ettei kehystä ole syytä suositella verkkosovellusten kehittämiseen, mikäli halutaan ottaa huomioon sovelluksen MVC-arkkitehtuurin oikeaoppisesta toteutuksesta aiheutuvien sovelluskehyksen kehitykseen ja ylläpitoon liittyvien laatuominaisuuksien edut sovelluksen suunnittelussa, toteutuksessa ja mahdollisessa jatkokehitystyössä.

Yhtä kaikki on tärkeää huomata, etteivät läpikäydyt tekniikat yksinään takaa toteutettavan verkkosivuston korkeaa laatua. Jos kehittäjä kehyksen suosituksista ja valmiina tulevista toimintamalleista huolimatta tekee esimerkiksi ristiinviittauksia sovelluksen MVC-arkkitehtuurin ohjaimen ja näkymän välillä, koodin siirrettävyys kärsii. Tällöin myös sovelluksen osien käyttö muissa projekteissa vaikeutuu.

Niin ikään kehyksen käytöstä huolimatta on mahdollista, että sovelluksen muunneltavuus kärsii: Jos kehittäjältä puuttuu selkeä tavoite koodin eri osa-alueiden sijoittamisesta suhteessa muuhun ohjelmakoodiin, voi myöhemmin sovellusta muokkaavilla kehittäjillä ilmetä vaikeuksia ohjelman rakenteen hahmottamisessa, kun koodin eri osat eivät ole löydettävissä selvän intuition avulla. Tämän takia kehittäjän on otettava selvää, millaiseen rakenteelliseen viitekehykseen sovellus tulisi kyseistä sovelluskehystä käytettäessä rakentaa.

Pelkkä hyväksi havaittujen suunnittelumallien seuraaminen ei johda onnistuneeseen toteutukseen. Kehittäjällä on oltava myös harjaannusta modulaarisen ja geneerisen koodin tuottamisesta. Lisäksi kehittäjällä on luonnollisesti oltava myös käyttöliittymän asetteluun ja ulkoasuun liittyvää osaamista sekä luontaista taipumusta havaita sovelluksen käyttöliittymään liittyviä epäkohtia.

6. Yhteenveto

Sovelluskehyksen valinta vaikuttaa paljon sovelluksen lopputulokseen. Vaikka loppukäyttäjän näkemä sovelluksen ulkomuoto saattaa eri sovelluksilla toteutettuna olla yhtenevä, voi sovelluskehyksen valinta vaikuttaa oleellisesti sovelluksen sisäiseen arkkitehtuuriin. Arkkitehtuuri taas vaikuttaa muun muassa siihen, kuinka helposti sovellukseen voidaan tehdä myöhemmin muutoksia.

Keskeisimmät tutkielman perusteella tehtävät havainnot liittyvät eri sovelluskehysten arkkitehtuurien eroihin. Se, mikä kehys on mihinkin käyttötarkoitukseen optimaalisin, riippuu pitkälti kehittäjän teknisestä taustasta, tottumuksista. Sopivan kehyksen valinnalla on kuitenkin vaikutuksia sovelluksen laatuominaisuuksiin. Vaikka sovellus voi näyttää ulkoisesti samalta millä tahansa kehyksellä toteuttaen, vaikuttaa kehyksen valinta sovelluksen sisäisen arkkitehtuurin kautta ohjelman ylläpidettävyyteen, laajennettavuuteen, tehokkuuteen ja koodin luettavuuteen. Koska sovelluksen tilanneen asiakkaan intresseihin ei useinkaan kuulu sovelluksen sisäiseen arkkitehtuuriin liittyvät yksityiskohdat, on sovelluksen kehittäjällä erityinen vastuu arkkitehtuurin valinnassa sekä siitä johtuvien laatuominaisuuksien esiintyvyydessä.

Tutkielman pohjalta havaitaan kolme keskeistä sopivan sovelluskehyksen valintaan liittyvää kriteeriä. Pidettäessä löyhää sovelluskomponenttien sidoksellisuutta tärkeänä ominaisuutena tulee valita sovelluksen pohjaksi kehys, jossa MVC-osien eriytys on viety mahdollisimman pitkälle. Tämän tutkielman tarkastelun kohteena olleista kehyksistä tällainen olisi Maria-kehys. Jos taas sovelluksen kehitystyössä kehitettävien komponenttien uudelleenkäytöllä ei ole ratkaisevaa merkitystä, vaan sovellus halutaan tehdä sovelluksen asettaman ongelmaviitekehyksen näkökulmasta yksittäisluontoisena sovellusratkaisuna, voidaan kallistua KnockoutJS-kehyksen käyttöön. KnockoutJS-kehyksen käyttöä voitaneen suositella myös tapauksissa, joissa käyttöliittymätapahtumien liittämisestä käyttöliittymään HTML-rakenteen seassa ei ole ongelma, vaan tärkeämpänä pidetään mahdollisimman nopeata sovelluksen kehitystyön etenemistä. Kolmas havainto liittyy jQueryMX-kehyksen käyttöön. Kehyksen käyttöä ei voitane suositella sovelluksen pohjana käytettävänä arkkitehtuuriratkaisuna, mikäli MVC-arkkitehtuurin suomien sidoksellisuutta vähentävien, ylläpidettävyyttä lisäävien sekä kehittäjäkunnan samanaikaisen yhteistyön mahdollistavien ominaisuuksien toteutuminen sovelluksen

kehitystyössä on tärkeää. Tältä pohjalta voidaan todeta, että jQueryMX sopii vain yksittäisten sovelluskehittäjien sovellustarpeisiin ja vain silloin, kun kehitetään yksinkertaisia verkkosovelluksia.

Havaittiin myös, että Javascriptin puutteellisen oliomallinnustuen vuoksi Javascript-sovelluskehysten kehittäjät ovat kukin pyrkineet luomaan omanlaisen tapansa toteuttaa oliomaailman käsitteet. Tämän vuoksi tiettyä sovelluskehystä käyttävä sovellus sidotaan tiukemmin vain sovelluskehyksen alaisuudessa toimivaksi. Kehysriippumattomia oliomallinnuskirjastoja on olemassa Javascriptille, mutta niitä käytetään sovelluskehyksissä vähän. Yleisen oliomallinnuskirjaston laajempi käyttö eri sovelluskehyksissä saattaisi helpottaa koodin siirrettävyyttä kehyksestä toiseen. Toisaalta on ymmärrettävää, että kehysten kehittäjät haluavat minimoida ulkopuolisten kirjastoriippuvuuksien määrän.

Yhtä kaikki, vertailemieni sovelluskehysten pyrkimys oliopohjaisiin ratkaisuihin sovelluksen rakenteen muodostamisessa mahdollistaa paremmin ylläpidettävien ja mukautuvien verkkosovelluksien luonnin. Useampien kirjastojen yhtäaikainen käyttö projektissa helpottuu, kun oliototeutuksen ansiosta yleiseen nimiavaruuteen asetettavien muuttujien ja metodien määrä vähenee.

Verkkosovelluksista tulee olioparadigman käytön ansiosta enemmän nykyisten ohjelmointikäytäntöjen mukainen, kun käytettävissä ovat muun muassa luokkien perintä- ja kapselointitoiminnallisuudet. Perinnän avulla luokan toimintaa voidaan muuttaa, säilyttäen yhteensopivuus luokkaa käyttävien muiden ohjelman osien kanssa. Kapseloinnilla luokan sisäiset toiminnot ja jäsenmuuttujat saadaan kätkettyä luokan ulkoisen rajapinnan ulottumattomiin.

Kun Javascript-kieltä alettiin käyttää verkkodokumenttien luomiseen 1990-luvulla, ei varmaankaan osattu kuvitella, kuinka paljon sen käyttö tulevaisuuden sovelluskehityksessä merkitsee. Nykyään verkkosovelluksetkin näyttävät nojautuvan vahvasti luokkapohjaisiin käsitteisiin, mikä on poikkeuksellista prototyyppipohjaisen kielen yhteydessä. Kuitenkin vasta äskettäin, vuodesta 2008, on alettu suunnitella enemmän nykyisten ohjelmointiperiaatteiden mukaista, täysin uudenlaista versiota Javascript-kielestä, nimeltään *EcmaScript Harmony* [ECMA-262, 2011; Resig, 2008]. Kieli sisältää paremman tuen luokille, perinnälle, jäsenmuuttujien ja -metodien näkyvyydelle sekä lukuisille yleisesti tunnetuille tietorakenteille kuten linkitetyille listoille.

Koska mikään selain ei vielä tue tätä uutta Javascriptin versiota, joutuu jokainen sovelluskehys vastaisuudessakin ratkaisemaan itse, kuinka mallintaa arkkitehtuurissaan oliomaailman peruskäsitteet. On myös selvää, etteivät verkkosivustot ja -sovellukset voi vielä pitkään aikaan nojautua pelkästään uuden Javascript-version syntaksiin, sillä verkkokäyttäjäkunnan selainkannalla on tapana päivittyä uusimpiin selainversioihin hitaasti.

Koin ongelmana sovelluskehysten vertailussa sen, kuinka saisin vertailutuloksista mahdollisimman täsmällisiä ja objektiivisia. Ongelmallista oli myös löytää sopivat kriteerit vertailtujen kehysten valintaan. Kuinka selvittää, minkä tyyppiset koodiesimerkit tuovat parhaiten esiin eri kehysten välisiä arkkitehtuurillisia etuja ja haittoja?

Vaihtoehtoisena lähestymistapana olisi voinut olla kvantitatiivinen vertailu. Numerodatan perusteella vertailun lopputulos olisi ollut ainakin näennäisesti objektiivisempi. Toisaalta kvantitatiivisen vertailun myötä moni sovelluskehysten arkkitehtuuriin liittyvä keskeinen ominaispiirre olisi voinut jäädä huomiotta.

Tämä tutkielma toi esiin verkkosovelluskehitykseen käytettyjen sovelluskehysten MVC-malliin liittyviä arkkitehtuurillisia puolia. Sovelluskehittäjät, jotka ovat tarkkoja sopivan arkkitehtuurin valinnasta, voivat hyötyä tämän tutkielman näkökohdista ja huomioista. On kuitenkin huomioitava, että sopivan sovelluskehyksen valinnassa on arkkitehtuurin oikeaoppisuuden lisäksi pohdittava, mitä muita mahdollisia toiminnallisuuksia ja työkaluja sovelluskehys tarjoaa kehittäjän käyttöön; rajaako valittu kehys kenties myöhemmässä vaiheessa esitettävien toiminnallisuusvaatimusten edellyttämien ulkopuolisten kirjastojen käyttömahdollisuuksia.

Lisäksi kaikki sovelluskehykset eivät tarjoa kaikkia mahdollisia verkkosovelluskehityksessä tarvittavia työkaluja. Toinen kehys voi esimerkiksi tarjota pitkälle kehittyneet rajapinnat hakukoneyhteensopivien Ajax-sivujen toteuttamiseen, kun taas toisessa nämä toiminnallisuudet joudutaan itse toteuttamaan tai vaihtoehtoisesti käyttämään neljännen osapuolen kirjastoa.

Yleisesti ottaen Javascript-pohjaisten verkkosovellusten tulevaisuus näyttää lupaavalta. Vahvimmat niistä saavat taakseen laajan yhteisön tuen, mikä edesauttaa niiden kehittymistä myös uusimpien Javascript-standardien mukaisiksi. Verkkopohjaiset sovellusratkaisut näyttävät tulevan yhä suositummiksi laajalla käyttäjärintamalla. Niin yritykset, voittoa tavoittelemattomat organisaatiot kuin julkisen sektorin sovellustarpeet tyydytetään yhä enemmän verkkopohjaisina sovellusratkaisuina. Samalla verkkosovellusten ohjelmalogiikka siirtyy yhä enemmän palvelinpuolelta selaimella ajettavaksi.

Nämä seikat yhdessä vahvistavat entisestään käsitystä kehittyneiden ja ajanmukaisten verkkosovelluskehysten tarpeellisuudesta. Parhaimmillaan verkkosovelluskehykset auttavat sovelluskehittäjiä luomaan organisoidumpaa ohjelmakoodia sekä nopeuttamaan valmiiden ratkaisujen tarjonnan avulla sovelluksen julkaisua.

Tutkielma jättää tilaa verkkosovelluskehysten jatkotutkimukselle. Koska selainpohjaiset ratkaisut yleistyvät nopeasti, myös tieteelliselle tutkimukselle on tarvetta. Tarpeellista olisi tutkia tarkemmin, miten MVC-arkkitehtuuria voitaisiin mukauttaa soveltumaan paremmin selainpohjaisten sovellusten asettamiin arkkitehtuurillisiin erityisvaatimuksiin.

Viiteluettelo

[Abbate, 2002] Janet Abbate, *About Adele Goldberg*, 2002. http://ieeeghn.org/wiki/index.php/Oral-History:Adele_Goldberg (Retrieved on 18.3.2014).

[Adobe, 2014] Adobe, *Creating your first desktop AIR application with the Flex SDK*, 2014. http://help.adobe.com/en_US/air/build/WS144092a96ffef7cc4c0afd1212601c9a36f-8000.html (Retrieved on 17.3.2014).

[Agile, 2013] Agile Support Team, *Performance Tuning Ajax based applications: Best practices*, 2013. http://www.agileload.com/agileload/blog/2013/08/02/performance-tuning-ajax-based-applications-best-practices (Retrieved on 21.3.2013).

[Backbone.js, 2014] Jeremy Ashkenas, *Backbone Documentation*, 2014. http://backbonejs.org/#View (Retrieved on 20.3.2014).

[Berners-Lee, 1989] Tim Berners-Lee, *World Wide Web: Proposal for a HyperText Project*, 1989. http://info.cern.ch/hypertext/WWW/Proposal.html (Retrieved on 17.3.2014).

[Botella et al., 2004] Pere Botella, Xavier Burgués, Juan-Pablo Carvallo, Xavier Franch, Gemma Grau, Jordi Marco, Carme Quer, ISO/IEC 9126 in practice: what do we need to know?. In: *Software Measurement European Forum 2004*, p. 297–306.

[Buse, 2008] Raymond Buse, A metric for software readability. In: *Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA '08)*, 2008, p. 121–130.

[Bychkov, 2013] Dmitriy Bychkov, *Desktop vs. Web Applications: A Deeper Look and Comparison*, http://www.seguetech.com/blog/2013/06/07/desktop-vs-web-applications-deeper-comparison (Retrieved on 17.3.2014).

[Classify.js, 2010] Pete Browne, *A Ruby-like Module & Class Inheritance Library*, 2010. http://classify.petebrowne.com/ (Retrieved on 20.3.2014).

[ECMA-262, 2011] *ECMAScript Language Specification.* http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (Retrieved on 7.3.2014).

[Fayad & Schmidt, 1997] Mohamed Fayad, Douglas C. Schmidt, Object-oriented application frameworks, 1997. *Comm. ACM*, 40(10), 32–38.

[Gamma et al., 1994] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Google Developers, 2013] *Chrome V8*, 2013. https://developers.google.com/v8/ (Retrieved on 14.3.2014).

[Haikala & Märijärvi, 2002] Ilkka Haikala, Jukka Märijärvi, *Ohjelmistotuotanto*. Talentum Oyj, 2002.

[Haneef, 1998] Nuzhat Haneef, Software Documentation and Readability: A Proposed Process Improvement. *ACM SIGSOFT Software Engineering Notes* 23(3), 1998, 75–77.

[Html5rocks.com, 2014] *HTML5 Features: File Access*, 2014. http://www.html5rocks.com/en/features/file_access (Retrieved on 17.3.2014).

[ISO-8402, 1994] International Organization for Standarization, *ISO Standard 8402: Quality management and quality assurance - Vocabulary*, International Organization for Standarization, 1994.

[ISO-9126, 2000] *ISO/IEC 9126-1, Information technology — Software product quality — Part 1: Quality model*, ISO/IEC, 2000.

[JavaEncrypt, 2011] *Obfuscating JavaScript with dirty code*, 2011. http://javaencrypt.com/ (Retrieved on 15.3.2014).

[JavaScriptMVC, 2014a] Justin Meyer, Brian Moschel, *JavaScriptMVC Documentation.* http://v32.javascriptmvc.com/docs.html#!jquerymx (Retrieved on 6.3.2014).

[JavaScriptMVC, 2014b] Justin Meyer, Brian Moschel, *JavaScriptMVC Controller.* http://v32.javascriptmvc.com/docs.html#!jQuery.Controller (Retrieved on 5.5.2014).

[JavaScriptMVC, 2014c] Justin Meyer, Brian Moschel, *JavaScriptMVC MVC Controller*. http://v32.javascriptmvc.com/docs.html#!mvc.controller (Retrieved on 5.5.2014).

[Jobs, 2010] Steve Jobs, *Thoughts on Flash*, 2010. http://www.apple.com/hotnews/thoughts-on-flash/ (Retrieved on 17.3.2014).

[Jørgensen, 1999] Magne Jørgensen, Software quality measurement. *Advances in Engineering Software* 30, 1999, 907–912.

[jQuery, 2014] The jQuery Foundation, *jQuery API documentation.* http://api.jquery.com/ (Retrieved on 28.2.2014).

[JSON, 2014] *Introducing JSON.* http://www.json.org/ (Retrieved on 28.2.2014).

[Kaye, 2003] Doug Kaye, *Loosely Coupled: The Missing Pieces of Web Services*, RDS Press, 2003.

[Kerr, 2013] Dean Kerr, *JavaScript MVC Frameworks*, 2013. http://scottlogic.com/blog/2013/12/06/JavaScript-MVC-frameworks.html (Retrieved on 15.3.2014).

[Leff & Rayfield, 2001] Avraham Leff, James Rayfield, Web-Application Development Using the ModelViewlController Design Pattern. In: *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing (EDOC '01)*, 2001, p. 118–127.

[lesscss.org, 2014] Alexis Sellier, Dmitry Fadeyev, *An overview of Less, how to download and use, examples and more*, 2014. http://lesscss.org (Retrieved on 17.3.2014).

[Loudon, 2010] Kyle London, *Developing Large Web Applications.* O'Reilly Media, 2010.

[Manoj, 2012a] Manoj Jaggavarapu, *Presentation Patterns: MVC, MVP, PM, MVVM*, 2012. http://manojjaggavarapu.wordpress.com/2012/05/02/presentation-patterns-mvc-mvp-pm-mvvm/ (Retrieved on 6.3.2014).

[Manoj, 2012b] Manoj Jaggavarapu, *MVP – Passive View*, 2012. http://manojjaggavarapu.files.wordpress.com/2012/05/mvp-passiveview.png (Retrieved on 6.3.2014).

[Mehran, 2010] Mehran Nikoo, *MVVM pattern.* http://mnikoo.files.wordpress.com/2010/09/slide5.png (Retrieved on 6.3.2014).

[Mesbah & van Deursen, 2007] Ali Mesbah, Arie van Deursen, Migrating Multi-page Web Applications to Single-page AJAX Interfaces. *Software Maintenance and Reengineering*, 2007, 181–190.

[Michaux, 2013] Peter Michaux, *Maria Framework*. http://peter.michaux.ca/maria/ (Retrieved on 1.11.2013).

[Microsoft, 2014] Microsoft, *JScript (ECMAScript3)*, 2014. http://msdn.microsoft.com/en-us/library/hbxc2t98(v=vs.85).aspx (Retrieved on 14.3.2014).

[Mozilla, 2014a] Mozilla, *Using files from web applications*, 2014. https://developer.mozilla.org/en-US/docs/Using_files_from_web_applications (Retrieved on 17.3.2014).

[Mozilla, 2014b] Mozilla, *JavaScript Reference: Object.create()*, 2014. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create (Retrieved on 21.3.2014).

[MVC-Process, 2010] RegisFrey, *A typical collaboration of the MVC components,* 2010. http://en.wikipedia.org/wiki/File:MVC-Process.svg (Retrieved on 25.4.2014).

[Observer, 2010] WikiSolved, *Observer pattern,* 2010. http://en.wikipedia.org/wiki/File:Observer.svg (Retrieved on 25.4.2014).

[Osmani, 2012a] Addy Osmani, *Journey Through The JavaScript MVC Jungle.* http://coding.smashingmagazine.com/2012/07/27/journey-through-the-javascript-mvc-jungle/ (Retrieved on 13.11.2013).

[Osmani, 2012b] Addy Osmani, *Learning JavaScript Design Patterns.* Volume 1.5.2. http://addyosmani.com/resources/essentialjsdesignpatterns/book (Retrieved on 13.11.2013).

[Osmani, 2012c] Addy Osmani, *Understanding MVVM – A Guide for Javascript Developers*. http://addyosmani.com/blog/understanding-mvvm-a-guide-for-javascript-developers/ (Retrieved on 24.4.2014).

[Peltomäki & Nykänen, 2006] Juha Peltomäki, Ossi Nykänen, *Web-selain-ohjelmointi*. WSOY, 2006.

[Purdy & Richter, 2002] Doug Purdy, Jeffrey Richter. *Exploring the Observer Design Pattern*, 2002. http://msdn.microsoft.com/en-us/library/ee817669.aspx (5.5.2014).

[Radziwill, 2008] Nicole Radziwill, *How ISO 8402 (9000 para 3.1.5) Relates Quality to Innovation*. http://qualityandinnovation.com/2008/10/22/how-iso-8402-relates-quality-to-innovation/ (Retrieved on 14.5.2014).

[Reenskaug, 1979] Trygve Reenskaug, *Models – views – controllers.* http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf (Retrieved on 6.3.2014).

[Reenskaug, 2014] Trygve Reenskaug, *MVC: XEROX PARC 1978–79.* http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html (Retrieved on 6.3.2014).

[Resig, 2008] John Resig, *ECMAScript Harmony.* http://ejohn.org/blog/ecmascript-harmony/ (Retrieved on 7.3.2014).

[RFC2616, 1999] Roy Thomas Fielding, Jim Gettys, Henrik Frystyk Nielsen, Larry Masinter, Paul Leach, Tim Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*. https://tools.ietf.org/html/rfc2616 (Retrieved on 20.3.2014).

[Ring.js, 2013] Nicolas Vanhoren, *Ring.js, JavaScript Class System with Multiple Inheritance*, 2013. http://ringjs.neoname.eu/ (Retrieved on 20.3.2014).

[Robbins, 2011] Charlie Robbins, *Scaling Isomorphic Javascript Code.* http://blog.nodejitsu.com/scaling-isomorphic-javascript-code (Retrieved on 28.2.2014).

[Ronan, 1996] Ronan Fitzpatrick, *Software Quality: Definitions and Strategic Issues*, Staffordshire University, 1996.

[RubyDoc, 2014] James Britt, *Ruby Standard Library Documentation*. http://www.ruby-doc.org/stdlib-2.1.1/libdoc/erb/rdoc/ (Retrieved on 6.3.2014).

[Sahgal, 2014a] Varoon Sahgal, *What is Javascript minification?*, 2014. http://www.programmerinterview.com/index.php/javascript/what-is-javascript-minification/ (Retrieved on 17.3.2014).

[Sahgal, 2014b] Varoon Sahgal, *What is the difference between minification and obfuscation?*, 2014. http://www.programmerinterview.com/index.php/javascript/minification-vs-obfuscation/ (Retrieved on 17.3.2014).

[Sass-lang.com, 2014] Hampton Catlin, Nathan Weizenbaum, Chris Eppstein, *CSS with superpowers*, 2014. http://sass-lang.com (Retrieved on 17.3.2014).

[Sommerville, 2007] Ian Sommerville, *Software Engineering*, Eight Edition, 2007.

[Sopyło, 2013] Maciej Sopyło, *Introduction to HTML5 Desktop Apps With Node-Webkit*, 2013. http://code.tutsplus.com/tutorials/introduction-to-html5-desktop-apps-with-node-webkit--net-36296 (Retrieved on 17.3.2014).

[Steidl et al., 2013] Daniela Steidl, Benjamin Hummel, Elmar Juergens, Quality Analysis of Source Code Comments. In: *Proceedings of 21st IEEE International Conference on Program Comprehension (ICPC)*, 2013, p. 83–92.

[Takada, 2014] Mikito Takada, *Single Page Apps in Depth*, 2014. http://singlepageappbook.com/ (Retrieved on 15.3.2014).

[Tan, 2012] Shin Hwei Tan, Darko Marinov, Lin Tan, Gary Leavens, @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In: *Proceedings of the 2012 IEEE 5th International Conference on Software Testing, Verification and Validation (ICST)*, 2012, p. 260–269.

[Tiirikainen, 2001] Vesa Tiirikainen, *Tietokone: Verkkotietokone tulee kiertotietä.* http://www.tietokone.fi/artikkelit/verkkotietokone_tulee_kiertotieta (Retrieved on 25.2.2014).

[Uhlig, 1995] Richard Uhlig, David Nagle, Trevor Mudge, Stuart Sechrest, Joel Emer, Instruction fetching: coping with code bloat. In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, 1995, p. 345–356.

[W3C, 1999] Dave Raggett, Arnaud Le Hors, Ian Jacobs, *HTML 4.01 Specification W3C Recommendation*, 1999. http://www.w3.org/TR/html401/ (Retrieved on 17.3.2014).

[W3C, 2008] Tim Bray, Jean Paoli, Michael Sperberg-McQueen, Eve Maler, François Yergeau, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 2008. http://www.w3.org/TR/REC-xml/ (Retrieved on 17.3.2014).

[W3C, 2011] Bert Bos, Tantek Çelik, Ian Hickson, Håkon Wium Lie, *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*, 2011. http://www.w3.org/TR/CSS2/ (Retrieved on 17.3.2014).

[Wirfs-Brock & Johnson, 1990] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener, *Designing Object-Oriented Software,* 1990. Englewood Cliffs, NJ: Prentice Hall. (Ch. 14)

[Yoon, 2014] Jason Yoon, *Software Metrics*, 2014. https://wiki.engr.illinois.edu/display/cs242sp14/Software+Metrics (Retrieved on 5.5.2014).